BUNDESREPUBLIK DEUTSCHLAND

CERTIFIED COPY OF
PRIORITY DOCUMENT

Prioritätsbescheinigung über die Einreichung einer Patentanmeldung

Aktenzeichen:	100 62 741.2
Anmeldetag:	15. Dezember 2000
Anmelder/Inhaber:	Siemens AG, München/DE
Bezeichnung:	Ablage von Projektinformationen mit XML
IPC:	G 06 F 17/50

Die angehefteten Stücke sind eine richtige und genaue Wiedergabe der ursprünglichen Unterlagen dieser Patentanmeldung.

München, den 7. November 2001
Deutsches Patent- und Markenamt
Der Präsident
Im Auftrag

Sieck

A 9161
03/00
EDV-L

Ablage von Projektinformationen in XML

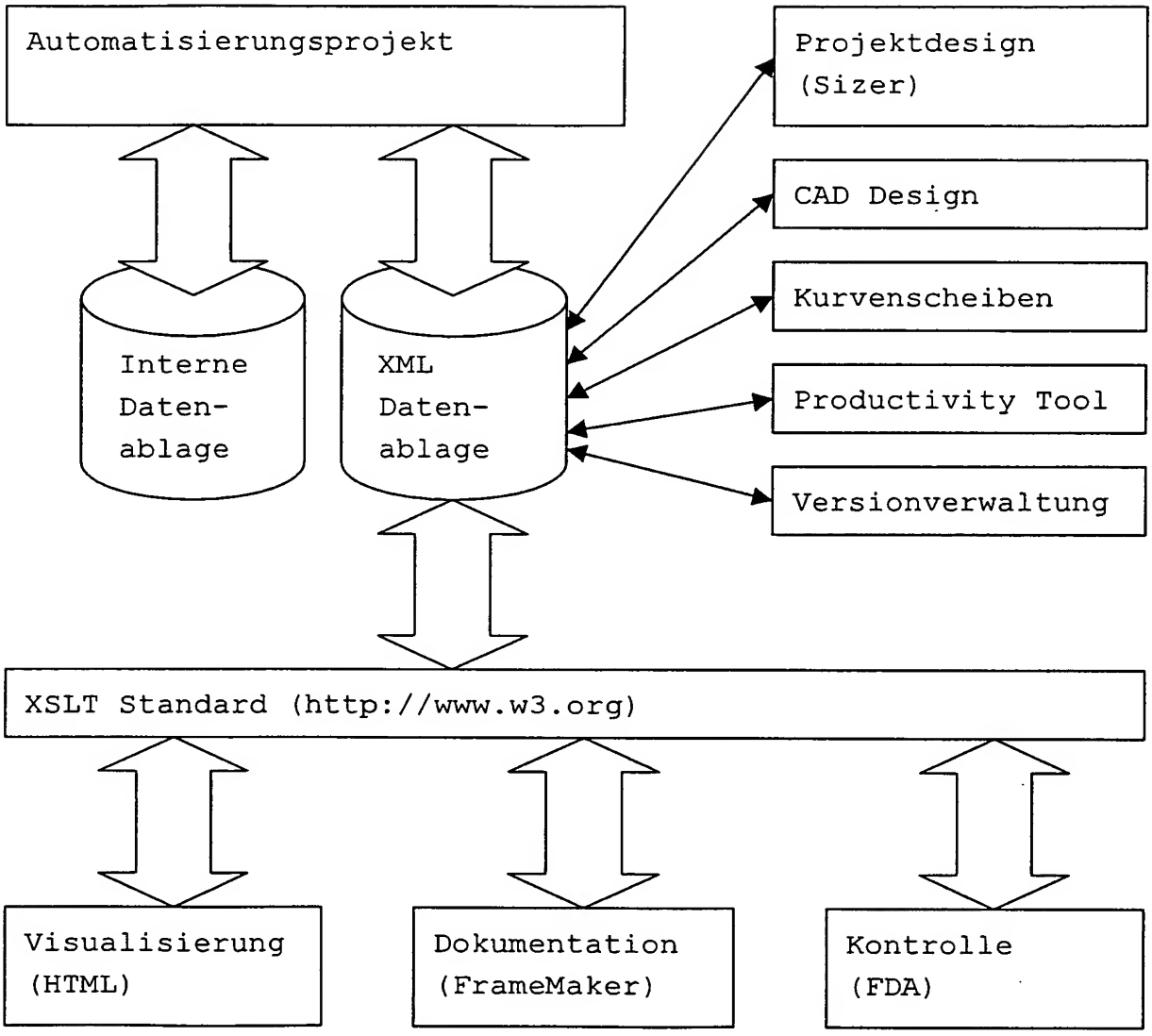
Die Ablage von Projektdaten einer Automatisierungskomponente erfolgt in XML. Dabei werden die Daten entweder direkt in diesem Format abgelegt oder entsprechende Export/Import Funktionen bereitgestellt um Projektdaten nach außen im XML Format bereitzustellen.
Durch dieses Vorgehen werden sind die Projektdaten außerhalb des Projekts in einem standardisierten Datenformat verfügbar. Auf dieser Basis können externe Werkzeuge für die vollständige oder teilweise weitere Verwertung oder Erzeugung von Projektdaten eingebunden werden.

Beispiele für solche externe Werkzeuge sind:

- Versionsverwaltungstools wie ClearCase oder PVCS
- Externe CAD-Programme
- Externe Kurvenscheibenwerkzeuge
- Externe Productivity Tools die Teile von Projektdaten generieren können
- Externe Skripting Mechanismen um die Erzeugung oder Duplizierung von Projektdaten zu unterstützen

Durch die Nutzung des Standardformats können kommerzielle Werkzeuge, die auf XML abbilden, genutzt werden.
Darüberhinaus können extern erzeugte auf XML basierenden Daten wie z.B. Maschinenbilder in das Projekt importiert werden.

Auf Basis der genannten Mechanismen können z.B. Änderungen von Projekten durch Vergleich der Projektdaten im XML-Format sehr einfach erkannt und Unterschiede feinstgranular aufbereitet werden. Dies ist insbesondere für FDA und dem hiermit notwendigen Change Control wichtig.


Automatisierungsprojekt
Projektdesign (Sizer)
CAD Design
Kurvenscheiben
Productivity Tool
Versionverwaltung
Interne Daten-ablage
XML Daten-ablage
XSLT Standard (http://www.w3.org)
Visualisierung (HTML)
Dokumentation (FrameMaker)
Kontrolle (FDA)

Einleitung

Der Entwicklungsname von SIMOTION war UMC (Universal Motion Control). Es stellt ein Basisautomatisierungssystem dar. Mit dieser Software ist es möglich durchgängige Automatisierungssysteme für Produktionsmaschinen zu erstellen (z.B. für Verpackungsmaschinen, Textilmaschinen, Pressen oder Kunststoffmaschinen). Die Anforderungen für die SIMOTION- Software basieren auf Erfahrungen mit anderen SIEMENS A&D - Produkten wie zum Beispiel SINUMERIK. Diese und andere Produkte des Bereiches A&D erfüllen schon heute die Anforderungen an das Automatisierungssystem. Aber die Gesamtlösung ist unzureichend und oft nicht wettbewerbsfähig, was sich mit der SIMOTION-Software ändern soll. Deswegen ist die Zielsetzung des SIMOTION, eine optimale Systemplattform für Automatisierungslösungen zu bieten. Außerdem soll SIMOTION die Möglichkeit bieten, Maschinenabläufe (SPS), spezifische Technologiefunktionen und vor allem Motion Control - Funktionen zu programmieren.

SIMOTION ist prinzipiell aus zwei Teilen aufgebaut.

Das Runtimesystem von SIMOTION, das zu Entwicklungszeiten UMC- RT hieß, ist auf drei verschiedenen Zielsystemen lauffähig:

- Drive-based,
- SPS-based und
- PC-based.

Es beinhaltet grundsätzlich ein Betriebssystem, Schnittstellen zum Antrieb, Basisfunktionen (wie Lageregler, Interpolator, Positionieren, Referenzieren, SPS-Funktionalität) und außerdem Treiber für den Feldbus PROFIBUS DP. Die Kommunikation des Gesamtsystems ist über eine Standardkommunikation mit SIMATIC, PC und OVA gewährleistet.
Technologiefunktionen können mit Hilfe von Firmwarekomponenten hinzugeladen werden. Das Konfigurieren dieser Funktiona-

lität wird auf einem Erstellsystem vorgenommen und durch einen Ladevorgang in das Zielsystem übertragen.

Das SIMOTION SCOUT hieß während der Entwicklung UMC- ES, was Erstell- bzw. Engineeringsystem bedeutete.
Es kann als eigenständiges System oder als System, das auf Step7 aufsetzt, genutzt werden.
Das SIMOTION SCOUT enthält eine Vielzahl von Funktionalität:

- Es ist möglich ein System oder eine Komponente zu konfigurieren, zu programmieren und zu projektieren.
- Programmiertools für das Runtimesystem sind im SIMOTION SCOUT enthalten.
- Der Test und die Inbetriebnahme, sowie die Diagnose- und der Service kann durch die Software unterstützt werden.
- SIMOTION SCOUT ist mit einer Bibliotheksfunktion für Programme, Konfiguration (HW-, SW-Konfiguration) und Maschinen ausgestattet.
- SIMOTION SCOUT kommuniziert mit dem Basis ES, das die Datenhaltung steuert und für den Zugriff auf die Daten zuständig ist.
- Außerdem gibt es die Möglichkeit des Teleservice, der Ferndiagnose und der Objektverwaltung.

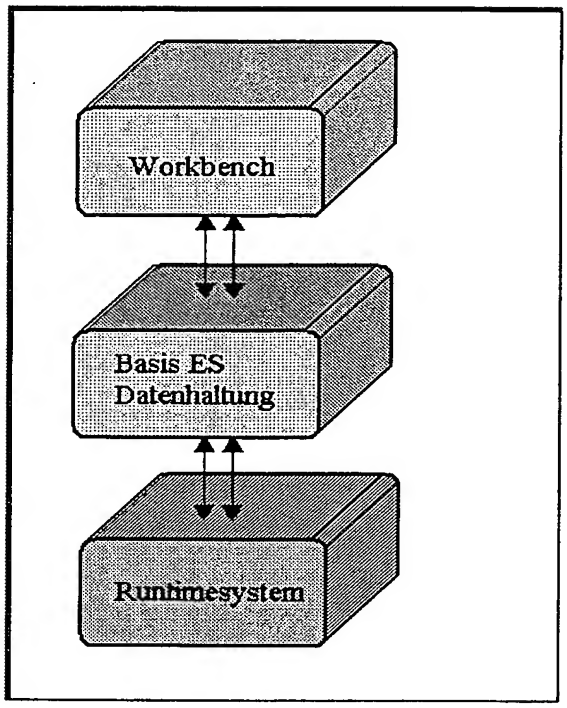


Abbildung 0.1: Das Universal Motion Control System

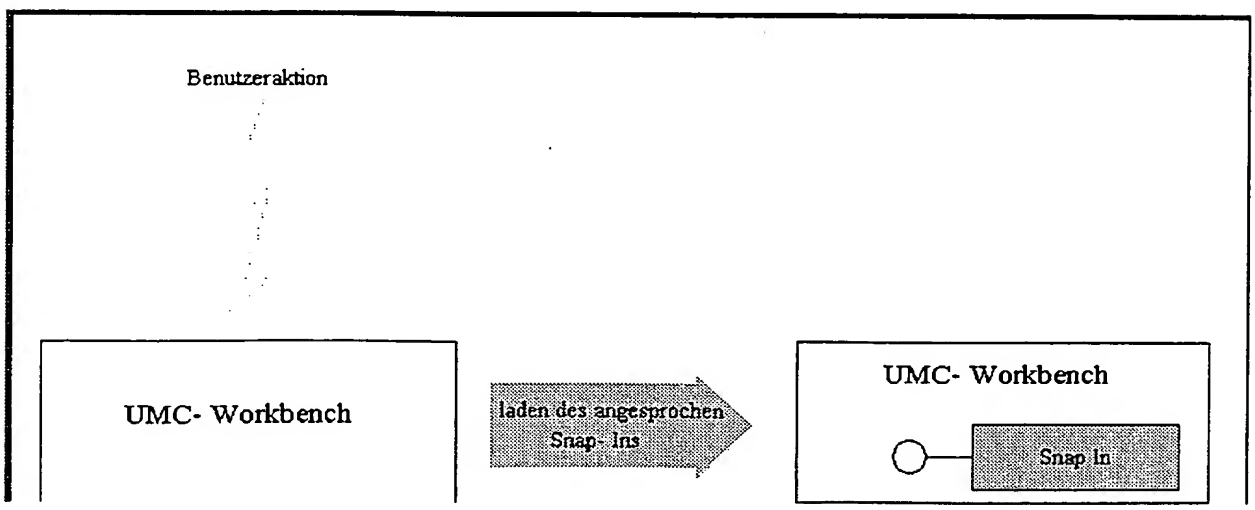
Die SIMOTION- Software ist nach dem *Three-Tier-Konzept* aufgebaut. Sie besitzt also eine grafische Oberfläche, eine Verarbeitungsschicht und eine Datenschicht.

Grundlagen

Dieses Kapitel soll einen Einblick in die Grundlagen der Software geben. Außerdem soll es alle im Export- Import- Mechanismus verwendeten Begriffe und Techniken vermitteln. Es stellt aber keine vollständige Dokumentation dar, deswegen verweise ich an geeigneter Stelle auf Referenzdokumente.

(a) Was ist ein Snap- In?

Allgemein versteht man unter einem Snap- In bei SIMOTION ein COM- Objekt. Die Workbench kann den Server in Form einer Dll laden und somit über die Schnittstellen des Servers auf Funktionen des Snap- Ins zugreifen. Die Kommunikation geht von einem Snap- In Host in der Workbench aus.
Neue Snap- Ins werden mit der Workbench installiert, können aber auch später hinzugefügt werden. Ebenso können Snap- Ins wieder entfernt werden. Wenn ein neues Snap- In hinzugefügt wird, legt es einen Registry- Eintrag an. Beim Starten der Workbench wird die Registry durchsucht. Neue Snap- Ins installiert die Workbench automatisch. Durch diesen Aufbau ist es möglich durch neue Snap- Ins dynamisch Funktionalität zu dem SIMOTION SCOUT hinzuzufügen und gegebenenfalls zu entfernen. Snap- Ins werden jedoch nicht sofort beim Start der Workbench geladen. Dies geschieht erst wenn eine Benutzteraktion ein Snap- In anspricht. Das Benutzterkommando wird von der Workbench an das entsprechende Snap- In weitergeleitet.



Beim Laden eines Snap- Ins kann die Oberfläche sowie die Menüstruktur von SIMOTION SCOUT geändert werden. Es ist möglich, dass Toolbars eingefügt werden und verschiedene Projektteile eine neue Darstellung erhalten.

Ein Snap- In muß bei SIMOTION bestimmten Anforderungen genügen:

- Da es ein COM- Objekt im herkömmlichen Sinne ist, muß die *IUnknown*- Schnittstelle implementiert sein, wie COM es vorschreibt.
- Für die Kommunikation zwischen der Workbench und dem Snap-In muß die Schnittstelle *IUMCSnapIn* vorhanden sein.
- Außerdem müssen alle Snap- Ins unter der Kategorie *„Snap-In of UMC-Workbench"* in der Registry eingetragen sein.

Wenn diese Anforderungen erfüllt sind, können noch optional neue Schnittstellen hinzugefügt werden. Diese Schnittstellen eignen sich, um spezielle Aufgaben zu übernehmen wie zum Beispiel Dateiverwaltung, Kommunikation mit anderen Snap- Ins oder grafische Schnittstellen.

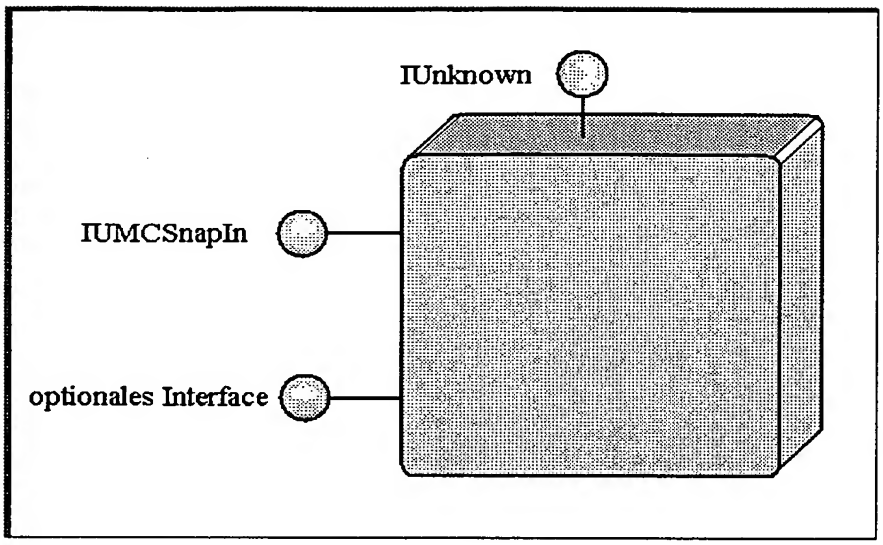


Abbildung 0.2.: Schnittstellen eines Snap- Ins

(b) Anforderungen und Vorteile des Export- Import- Tools

Durch die Export- Import- Software soll es ermöglicht werden, dass die Daten eines SIMOTION- Projektes aus dem Basis ES in ein lesbares ASCII- Format exportiert werden können. Mit Hilfe dieser exportierter Daten soll durch die Import- Funktion ein lauffähiges Projekt erzeugt werden können.

Die exportierten Daten werden im XML- Format gespeichert. Dabei wird unter den Teilkomponenten eines Projektes unterschieden, indem sie in verschiedenen XML- Dateien beschrieben werden. Die bei dem Export entstandenen Dateien liegen in einer Verzeichnisstruktur vor, die an den Aufbau des SIMOTION-Projektes angelehnt ist. An die Export- Import- Software des SIMOTION SCOUT werden folgende Anforderungen gestellt:

- Die entstandenen Dateien sollen die Projektdaten mit Hilfe einer XSL- Datei in einer geordneten, tabellarischen Form im Internet- Explorer 5 anzeigen.
- Exportierte XML- Dateien sollen bei der Diagnose eines Projektes hilfreich sein. Dies wird durch die Navigation über das Projekt im Internet- Explorer erreicht.

- Weiterhin sollen durch einen modularen Aufbau des exportierten Projektes einzelne Teilkomponenten eines Projektes komfortabel ausgetauscht und unabhängig von der Version der SIMOTION- Software in andere Projekte eingefügt werden können.
- Durch einen strukturierten Aufbau der Dateien im Verzeichnisbaum, soll die Archivierung und Erstellung von Versionen eines SIMOTION- Projektes vereinfacht werden.
- Außerdem kann man Dateien über das Internet, per Email oder über FAX an dritte weiter zu geben.
- Es ist prinzipiell möglich, dass der Benutzter mit Hilfe eines ASCII- Editor Daten des Projektes ändern kann und diese durch den Import in das Projekt übernehmen kann. Wobei die Änderung von Projektdaten normalerweise im Projekt in der Workbench vorgenommen werden sollte.

(c) Einordnung des Exports und des Imports in die SIMOTION- Software

Das in der Einleitung genannte *Three-Tier-Konzept* kommt auch in der Import- Export- Software zur Anwendung.
Die grafische Benutzteroberfläche wird in der Workbench über ein Snap- In angeboten.
Die gesamte Verarbeitung des Ex- oder Imports wird durch den XML- Server realisiert. Er kann über Schnittstellen auf die Daten des Basis ES lesend oder schreibend zugreifen. Über einen sogenannten *Connection-Point* werden die Ausgaben bei dem Export oder Import an das Snap- In in der Workbench gegeben. Die Daten werden mit Hilfe eines XML- Parser übersetzt. Bei dem Export ist die Datenschicht das Basis ES, aus dem die Daten der XML- Files gewonnen werden. Bei dem Import ist dies umgekehrt. Die XML- Dateien werden ausgelesen und die Daten in das Basis ES geschrieben.

10

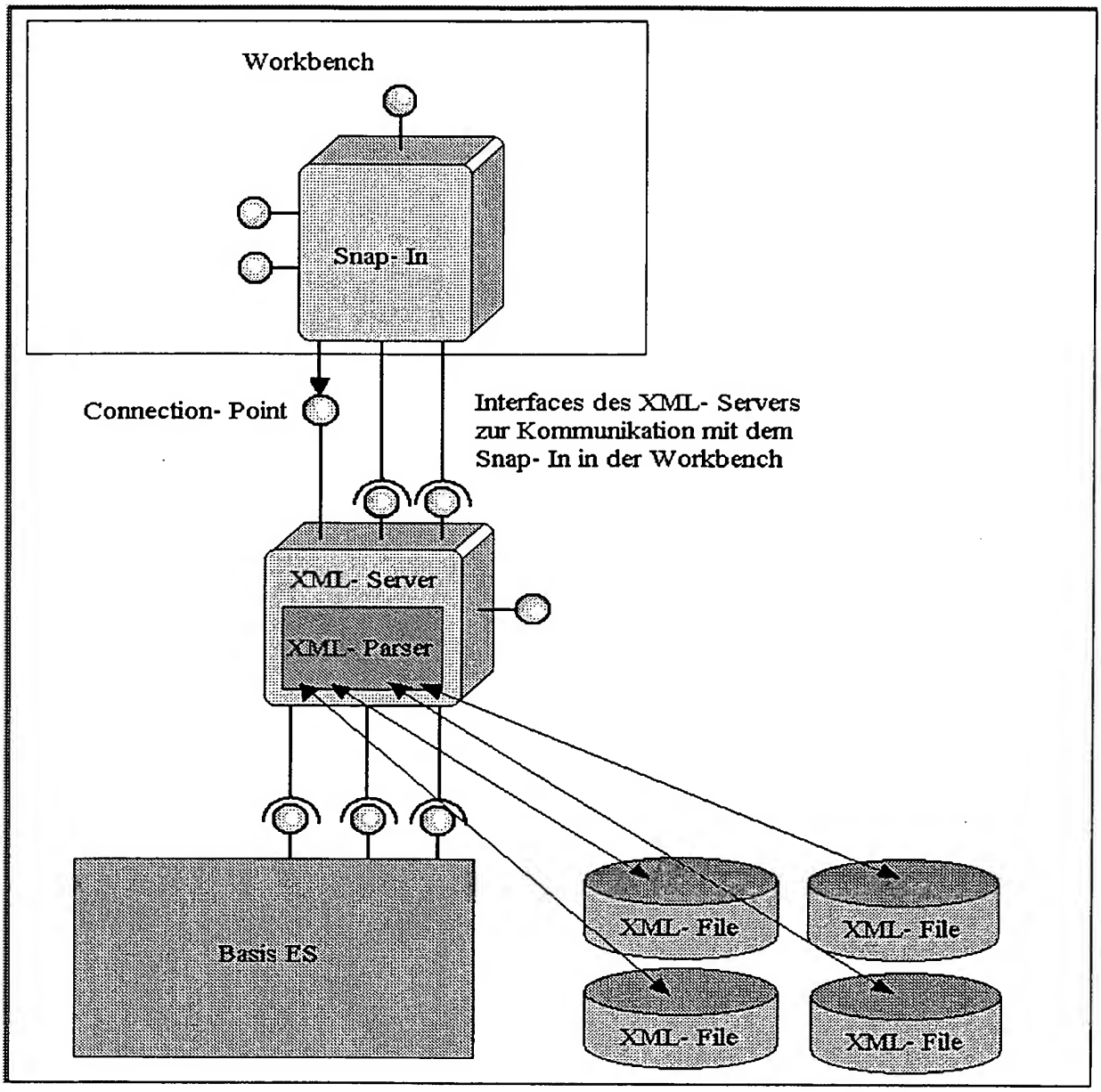


Abbildung 0.3.: Kommunikation des Export- Import- Tools

(d) Das XML- Format

XML (extensible Markup Language) ist eine Metasprache für das Definieren von Dokumenttypen. Es ist damit möglich, Dokumente zu erstellen, die alle den gleichen Grundmustern im Aufbau folgen. Auf der Grundlage dieser Dokumente können Programme leichter arbeiten.

Andere Vorteile von XML sind, dass die Daten strukturiert und hierarchisch beschrieben werden können. Außerdem ist die Sprache Plattform unabhängig und einfach von Mensch und Maschine zu erzeugen. Ein entscheidender Aspekt für die Verwendung von XML ist, dass eine strickte Trennung zwischen Inhalt, Darstellung und Struktur vorgenommen wird.

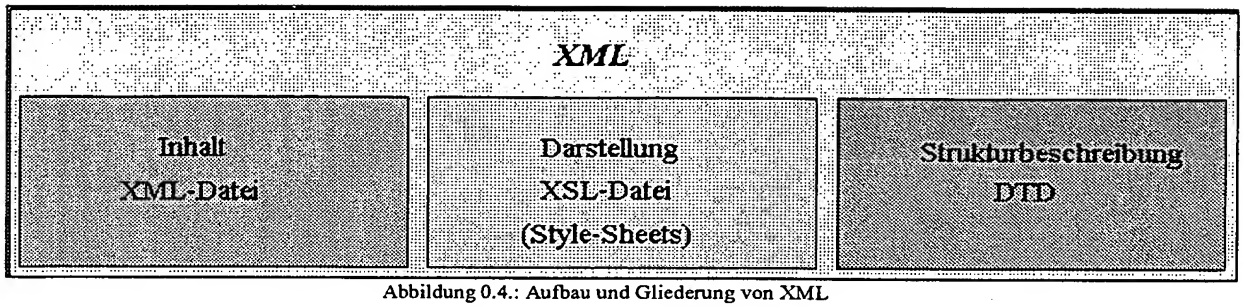


Abbildung 0.4.: Aufbau und Gliederung von XML

(e) Unterschiede zwischen XML und HTML

Warum die meisten Anbieter, Programmierer und Anwender von HTML auf XML umsteigen ist sehr einfach zu erklären. Während HTML eine Auszeichnungssprache ist, ist es mit XML möglich eine Auszeichnungssprache zu definieren.
Vorteilhaft ist auch das Definieren eigener Tags. XML- Files können ganz bestimmte Tags enthalten und diese können wiederum Tags enthalten. Dadurch kann ein XML- File einen Baum aufbauen, der zu einer strukturierten Trennung verschiedener Inhalte führt. Damit ist die Grundlage für eine maschinelle Bearbeitung des Dokumentes gewährleistet, die mit einem HTML-Dokument leider nicht gegeben ist. Ein großer Unterschied zwischen HTML und XML ist die Trennung von Inhalt, Darstellung und Struktur. Nachteilig erscheint manchen Programmierer, dass er bei HTML sehr frei in seiner Formulierung ist, während er bei XML auf die Wohlgeformtheit eines Dokumentes achten muß (siehe Kapitel (f)).

(f) Gültigkeit, Wohlgeformtheit, validierende und nicht-validierende Parser

Bei der Verarbeitung von XML- Dokumenten, wird zwischen zwei Instanzen unterschieden: dem XML- Parser und der Anwendung. Der Parser verrichtet seine Arbeit im Sinne einer Anwendung und ist in den meisten Fällen ein Browser. Es kann mit zwei verschiedenen Parsern gearbeitet werden, mit dem DOM- oder dem SAX- Parser. Der DOM- Parser erstellt einen XML- Baum und der SAX- Parser ist ereignisorientiert. Diese beiden Parser

werden im Kapitel 0 näher erklärt. Alle Parser sollten jedoch diese Eigenschaften erfüllen:

- Ein Parser prüft ein XML- Dokument auf Wohlgeformtheit und gibt gegebenenfalls Fehler aus.
- Wenn das Dokument aus mehreren Bestandteilen aufgebaut ist, werden diese zusammen gefügt.
- Außerdem wird ein Baum erstellt, durch den die Anwendung auf die Inhalte der Tags zugreifen kann.

Wenn ein Dokument auf Gültigkeit untersucht wird, vergleicht der Parser das Dokument mit der DTD. Da dieser Vergleich sehr umfangreich ist, wurde das Konzept der Wohlgeformtheit eingeführt. Bei diesem Konzept wird lediglich sichergestellt, dass die Erstellung des Baumes möglich ist.

Grundsätzlich ist ein Dokument wohlgeformt, wenn es folgende sechs Eigenschaften besitzt:

1. Jedes XML- Dokument muß mit einer Deklaration beginnen. Wie zum Beispiel: <?XML version = "1.0" ?>.
2. Alle Elemente müssen ein Ende- Tag besitzen:

   <title> Der Export - Import - Mechanismus des SIMOTION SCOUT </title>

   Ebenso müssen leere Elemente wie
 ein Ende- Tag besitzen:

   Entweder
 oder
 </BR>.
3. Alle Elemente sind *case-sensitive*, das heiß zwischen Groß- und Kleinschreibung wird unterschieden.
4. Alle Attributwerte sind in Hochkommas zu setzen.
5. Ein sauberes Verschachteln der Tags ineinander muß gewährleistet sein. Das heißt untergeordnete Tags müssen geschlossen werden bevor übergeordnete Tags geschlossen werden.
6. Ein Element muß als eine Art Container alle anderen Elemente einbetten.

Da bei der Prüfung auf Wohlgeformtheit kein Abgleich mit der DTD statt findet, ist ein wohlgeformtes XML- Dokument nicht unbedingt gültig.

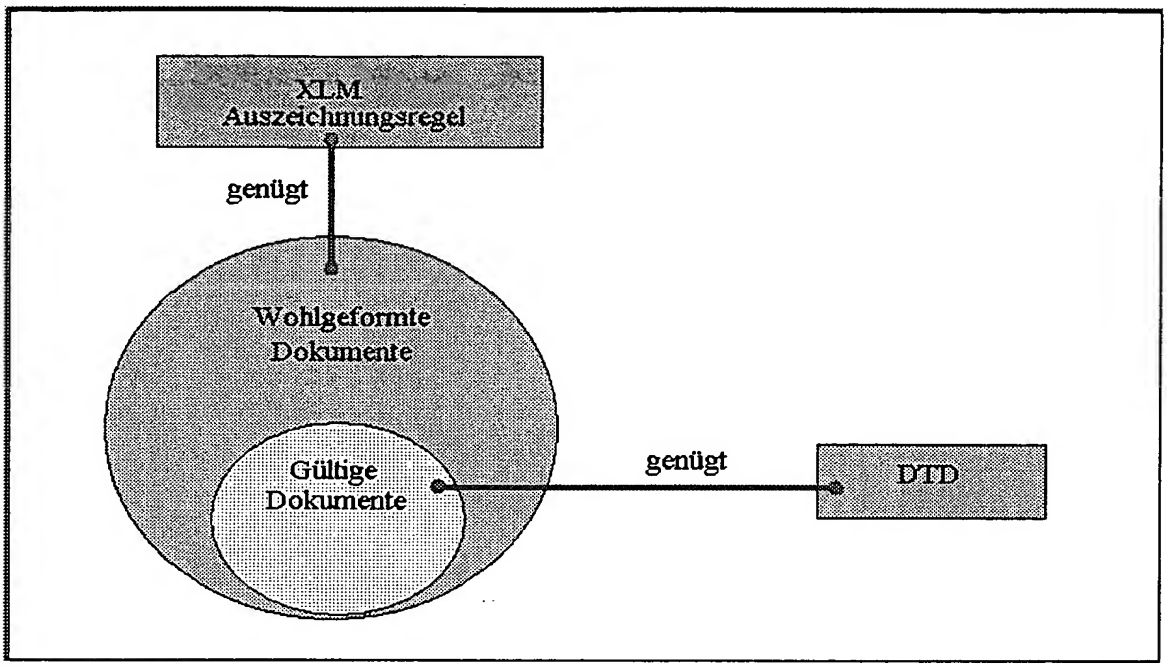


Abbildung 0.5.: Unterscheidung Wohlgeformtheit und Gültigkeit eines XML- Dokuments

Aus diesen Gründen kommen zwei verschiedene Parser zur Anwendung. Ein validierender Parser prüft auf Gültigkeit, ein nicht validierender Parser prüft auf Wohlgeformtheit eines XML- Dokuments. Wenn auf die Gültigkeit eines Dokumentes geprüft wird, ist die Prüfung auf Wohlgeformtheit eingeschlossen.

(g) Die Stylesheets (XSL)

XSL(extensible Stylesheet Language) ist eine Beschreibungssprache der Darstellung eines XML- Dokumentes. Mit Hilfe verschiedener XSL- Dateien können XML- Dokumente andere Darstellungen annehmen.
Bei HTML wird die Darstellung mit *cascading Stylesheet Language* (CSS) beschrieben. Diese Beschreibungsart kann ebenso in XML- Dokumenten eingesetzt werden, jedoch gibt es bei XSL einige Vorteile gegenüber CSS. XSL ist eine Programmiersprache im herkömmlichen Sinne, das heißt IF- Abfragen und Speicherung von Werten in Variablen können gesetzt werden. Einzelne

Komponenten können genommen werden und an anderen Plätzen auftauchen, wie zum Beispiel in der Kopf- oder Fußzeile. Da XSL nicht an die Sprachen der westlichen Welt angepasst ist und damit nicht ausschließlich horizontale Schriftzeichen dargestellt werden können, ist es global verwendbar.
Damit ein XML- Dokument eine XSL- Datei zur Darstellung heranzieht, muß diese am Anfang des XML- Dokumentes bekannt gemacht werden.
Zum Beispiel:

```
<?xml-stylesheet type="text/xsl" href="umc.xsl"?>
```

(h) Die Struktur einer XML- Seite (DTD)

In einer DTD(Documenttype- Definition) wird die Struktur und die Gliederung eines XML- Dokumentes festgelegt. Alle in einem XML- Dokument verwendeten Elemente müssen in der zugehörigen DTD beschrieben werden. Weiterhin muß in der DTD festgelegt werden, wie die Elemente verschachtelt sind und welche Werte die Attribute annehmen können.
Durch die Festlegung der Regeln für XML- Dokumente in einer DTD, ist es möglich einer Anwendung das Auslesen und die Verarbeitung der Daten zu überlassen.
DTD können in einer separaten Datei untergebracht werden (externe DTD). Anderenfalls kann die DTD am Anfang des XML- Dokumentes stehen, wobei sie hier interne DTD genannt wird.

```
Ein Ordner besteht aus mindestens einem Buch
<!ELEMENT        ordner          (book)+>
Der Ordner hat als Pflichattribut die ID
<!ATTLIST        ordner          id ID #REQUIRED>
Ein Buch besitzt einen Titel, mindestens einen Autor und wurde durch einen Verlag veröffentlicht
<!ELEMENT        book            (title, author+, puplisher)>

<!ATTLIST        book            isbn CDATA #REQUIRED>
Ein Titel besteht aus einem String
<!ELEMENT        title           (#PCDATA)>

<!ELEMENT        author          (#PCDATA)>

<!ELEMENT        publisher       (#PCDATA)>
```

Abbildung 0.6.: Beispiel für eine DTD

Das Snap- In in der Workbench

Das Snap- In des XML- Tools ist ein mittels ATL geschriebener COM- Server, der eine Schnittstelle zwischen dem Benutzer und dem XML- Server darstellt. Durch bestimmte Benutzeraktionen kann das Snap- In geladen werden. Über die grafische Oberfläche kann der Benutzer den Export und den Import bedienen. Die Kommunikation zwischen Snap- In und Workbench wird über verschiedene Schnittstellen des Snap- Ins ermöglicht.

(i) Das GUI des Snap- Ins

Dieses Kapitel beschreibt die grafische Oberfläche, die das Snap- In zur Verfügung stellt, und dessen Bedienung zum Ex- oder Import von Projektdaten.

(j) Starten des Ex- oder Imports

Das Snap- In wird beim Klicken auf den entsprechenden Button oder beim Auswählen des Exports oder Imports im Menü „Datei" geladen.

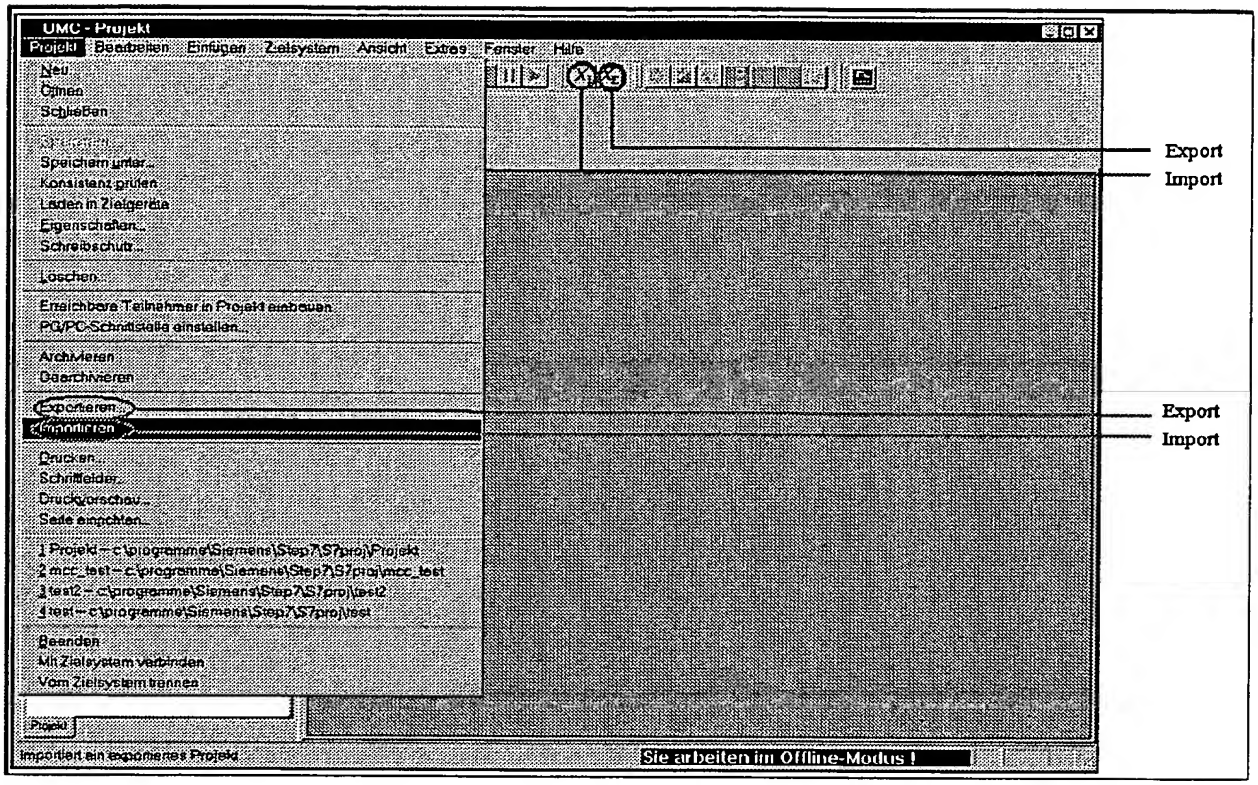


Abbildung 0.1.: Laden des Snap- Ins durch Benutzeraktionen

(k) Der Export eines Projektes

Der Export eines Projektes ist natürlich nur möglich, wenn ein Projekt geöffnet ist. Nach dem des Exports wird das Snap-In geladen und es erscheint dieser Dialog:

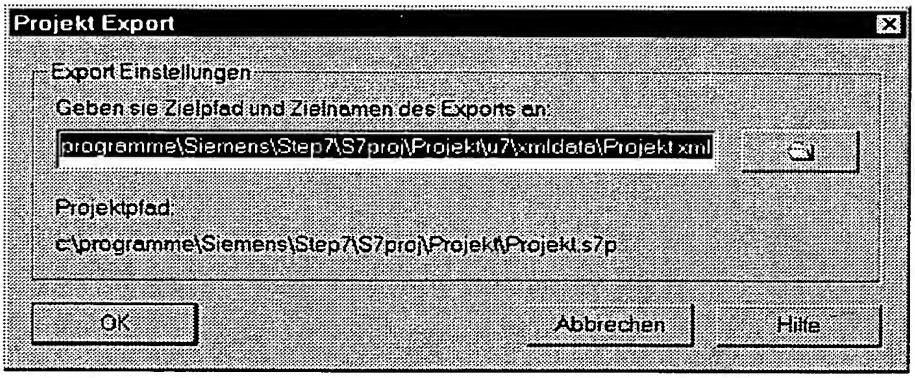


Abbildung 0.2.: Der Exportdialog

Der Projektpfad beschreibt das Verzeichnis, in dem die exportierten Daten gespeichert werden sollen. Man kann im Dialog einen Ordner angeben, der das exportierte Projekt beinhalten soll. Durch den „Ordner- Button" kann der Ordner aus dem Verzeichnisbaum ausgewählt werden.

Abbildung 0.3.: Wählen eines Verzeichnisses für das zu exportierende Projekt

Dabei ist zu beachten, dass alle im Ordner vorhandenen Dateien zu Beginn des Exports gelöscht werden. Mit dem OK- Button wird letztendlich der Export gestartet.

(I) Der Import eines Projektes

Der Import hat zwei Möglichkeiten ein Projekt zu importieren. Zum Einem können die Daten in ein bestehendes Projekt eingefügt werden. Dazu muß der Import gestartet werden, wenn ein Projekt geöffnet ist. Der Benutzer wird vor dem Import gefragt, ob er das Projekt ändern will. Wenn der Import in ein bestehendes Projekt durchgeführt wird, werden neue Daten eingefügt, Daten, die im zu importierenden Projekt sowie im bestehenden Projekt vorkommen, durch die importierten Daten aktualisiert und bestehende Daten beibehalten.
Zum Anderen kann beim Import ein neues Projekt angelegt werden. In diesem Fall muß der Import gestartet werden, wenn kein Projekt geöffnet ist. Das importierte Projekt ist nach dem Import mit dem exportierten Projekt identisch.
Der Import blendet einen ähnlichen Dialog auf wie der Export.

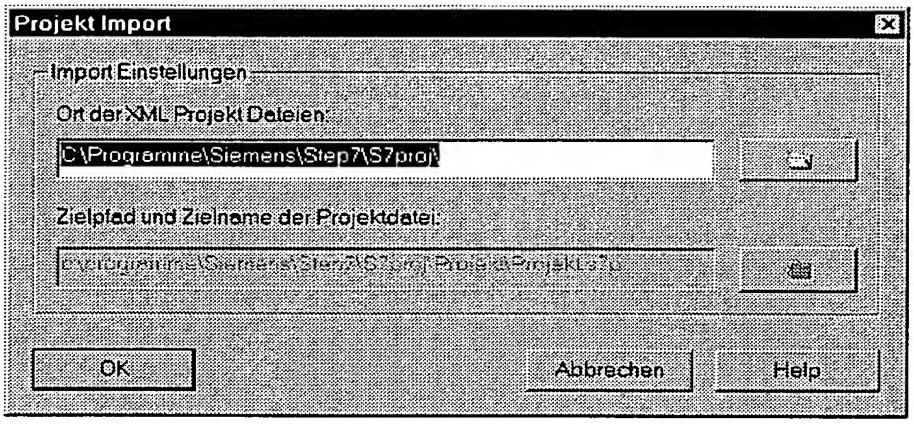


Abbildung 0.4.: Der Importdialog

Der Zielpfad zeigt auf das neu angelegte Projekt oder auf das Projekt, in das importiert werden soll. Hier ist ebenfalls der Projektpfad angegeben, in dem das Projekt importiert wird. Es sollte in diesem Dialog eine XML- Datei ausgewählt werden, die die Wurzel des zu importierenden Projektes ist. Diese XML- Datei kann durch einen File- Dialog ausgewählt werden, der durch den „Ordner- Button" angezeigt wird.

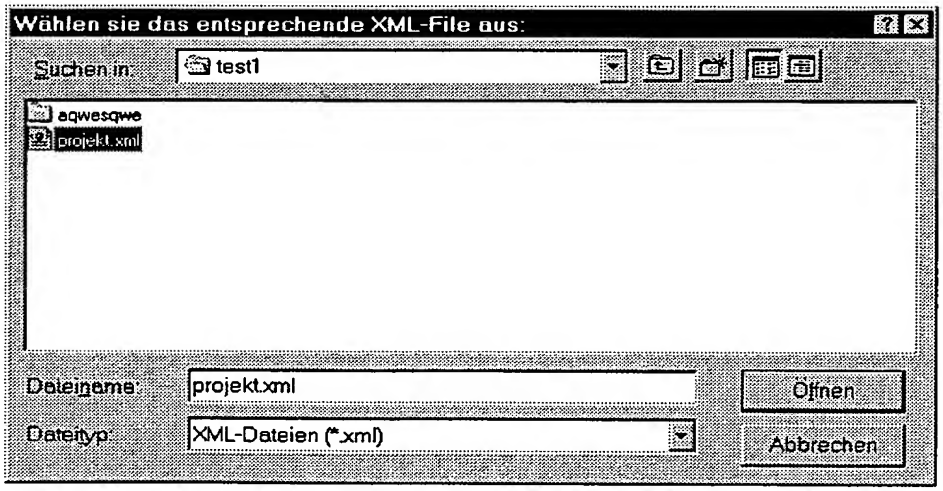


Abbildung 0.5.: Wählen einer XML- Datei für den Import

(m)Die Statusausgabe des Ex- bzw. Imports

Der Status des Exports oder des Imports wird jeweils in einem Ausgabefenster dokumentiert. Dort wird der Benutzer auf Feh-

ler, die rot hervorgehoben sind, aufmerksam gemacht, die Beendigung des Exports oder Imports ausgegeben und ein Link zu den XML- Dateien anzeigt.

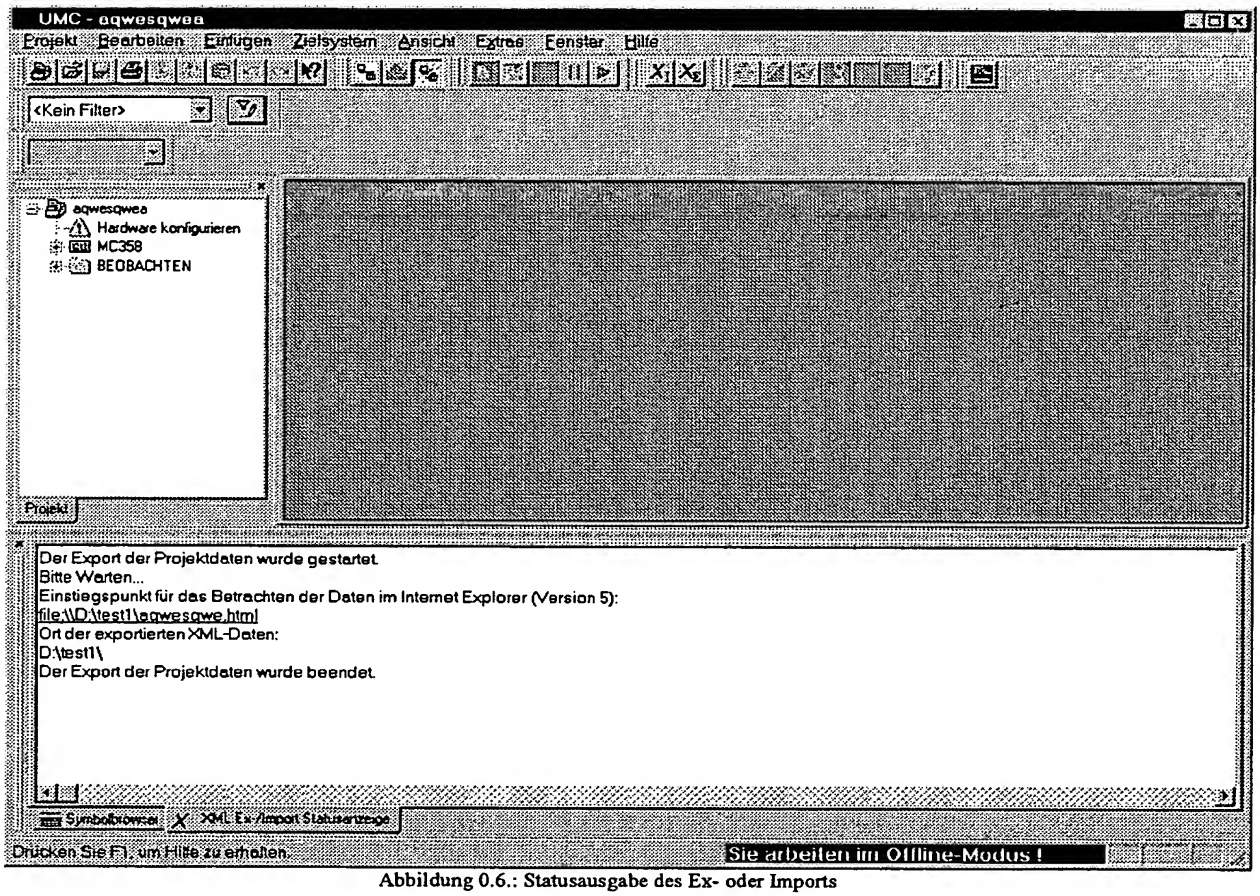


Abbildung 0.6.: Statusausgabe des Ex- oder Imports

Die gesamte Verarbeitung des Ex- oder Imports wird durch den XML- Server realisiert. Dies ist ein COM- Server, der das Auslesen und Übersetzten der Daten übernimmt. Bei dem Export ist die Datenschicht das Basis ES, aus dem die Daten der XML-Files gewonnen werden. Bei dem Import ist dies umgekehrt. Die XML- Dateien werden ausgelesen und die Daten ins Basis ES geschrieben.

(n) Die Interfaces des Snap- Ins

Die Schnittstellen eines Snap- Ins ermöglichen die Kommunikation zwischen der Workbench und dem Server des Snap- Ins. Ein Snap- In muß die Schnittstelle *IUMCSnapIn* implementieren, um als Snap- In erkannt zu werden. Die Schnittstellen *IUMCHelp* und *IUMCSnapInProperties* beinhalten verschiedene Funktionen,

die ein Snap- In verwenden kann. Natürlich können andere Snap- Ins diese Schnittstellen implementieren. Deswegen gehe ich auf diese Schnittstellen nur kurz ein und schenke der speziellen Schnittstelle des XML- Servers (*IXMLStatus*) mehr Aufmerksamkeit.

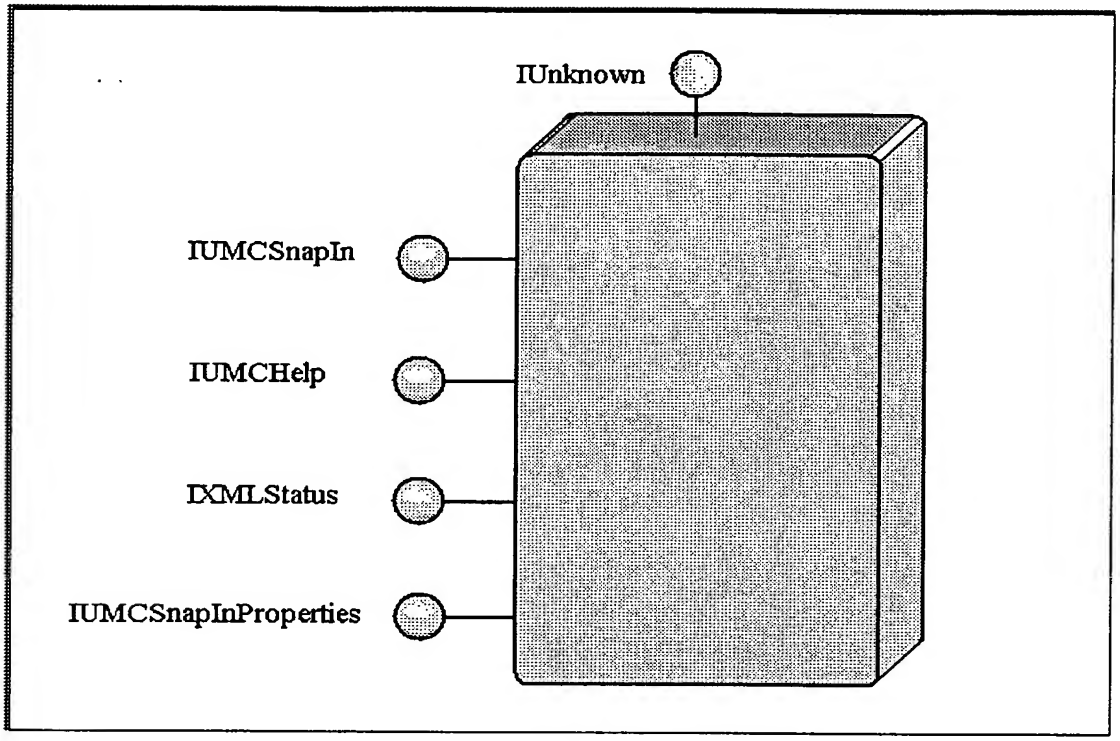


Abbildung 0.7.: Schnittstellen des XML- Snap- Ins

(o) IUMCSnapIn

Die Schnittstelle *IUMCSnapIn* ist die obligatorische Schnittstelle für Snap- Ins wie die *IUnknown* Schnittstelle von COM. Jedes Snap- In muß diese Schnittstelle implementieren wobei es unerheblich ist, ob das Snap- In eine grafische Oberfläche besitzt oder nicht. Weiterführendes Dokument zu diesem Interface ist unter \\erlf452a\umc all$\doc\Icd\ICD SnapIn\ICD SnapIn SnapInHost.doc zu finden.

(p) IUMCHelp

Um eine Hilfe zu einem Snap- In zu erhalten, ist es möglich mit F1 eine Hilfedatei zu öffnen. Diese Funktionalität wird durch das Interface *IUMCHelp* zur Verfügung gestellt. Zu dieser Schnittstelle ist eine Dokumentation unter

\\erlf452a\umc all$\Systemtest\Engineering System\Infos\Kochbuch ES\Kochbuch ES.doc vorhanden.

(q) IUMCSnapInProperties

Dieses Interface übermittelt der Workbench die Eigenschaften des Snap- Ins. Das ist nötig, da die Workbench sehr allgemein gehalten ist und keine spezifischen Daten des Snap- Ins kennt. Wenn ein Snap- In diese Schnittstelle implementiert, kann eine bestimmte Darstellung erreicht werden. Außerdem können Änderungen während der Laufzeit der Workbench übermittelt werden. Die Schnittstelle *IUMCSnapInProperties* ist in dem Designdokument \\erlf452a\umc all$\doc\Icd\ICD SnapIn\ICD SnapIn Properties.doc näher beschrieben.

(r) IXMLStatus

Diese Schnittstelle ermöglicht das Konvertieren eines Devices, wobei der Typ des Devices verändert wird. Die Konvertierung erfolgt durch das Exportieren der Devicedaten und anschließendes Importieren der Devicedaten. Damit hat diese Schnittstelle mit dem eigentlichen Export und Import eines Projektes nichts zu tun, verwendet aber einen ähnlichen Mechanismus, um ein Device zu konvertieren.
Die Schnittstelle besitzt zwei Funktionen: *ExportAndConvertESObject* und *ImportESObject*. *ExportAndConvertESObject* exportiert und konvertiert, wie der Name schon sagt, ein Device. Dieser Methode muß eine DeviceID, eine TypeID und ein Zielverzeichnis übergeben werden. Wenn das Device exportiert wird,

wird die alte TypeID durch die übergebene ersetzt. Alle Dialoge und Benutzeraktionen werden vernachlässigt, da der Export nur intern im Programm durchgeführt wird. Den eigentlichen Export eines Devices übernimmt der XML- Server, auf den ich im Kapitel 0 näher eingehen werde. Da in diesem Fall nur ein Teil des Projektes exportiert werden soll, wird eine Maske gesetzt, die das spezifiziert. Die Funktion *ExportAndConvertESObject* liefert nach erfolgreichem Export ein String des XML- Dateinamens zurück.

Die Methode *ImportESObject* importiert das Device und erhält den Ordner, der die exportierten Dateien enthält, und die XML- Datei, die bei *ExportAndConvertESObject* zurückgeliefert wurde. Bei dem Import ist es ebenso nicht notwendig die Dialoge anzuzeigen, da es sich wie beim Export um einen internen Import handelt. Nun importiert der XML- Server das Device auf Grund der übergebenen XML- Datei und gibt die DeviceID an die aufrufende Funktion zurück.

Der XML- Server

Der XML- Server ist ein ATL- Server und führt den eigentlichen Ex- oder Import der Daten durch. In diesem Kapitel werde ich auf die Funktionsweise und den Aufbau des Servers eingehen.
Die Dll wird in das Programm geladen, wenn die Schnittstelle des Servers angesprochen wird. Die Schnittstelle heißt *IXMLServer* und bieten zwei Methoden an. Mittels dieser Methoden wird der Import und der Export angestoßen, deswegen heißen sie *StartImport* und *StartExport*. Diese Funktionen sind die einzigen Funktionen, die von Außen aufgerufen werden können, denn wie aus dem Bild ersichtlich stehen keine weiteren Schnittstellen zur Verfügung.

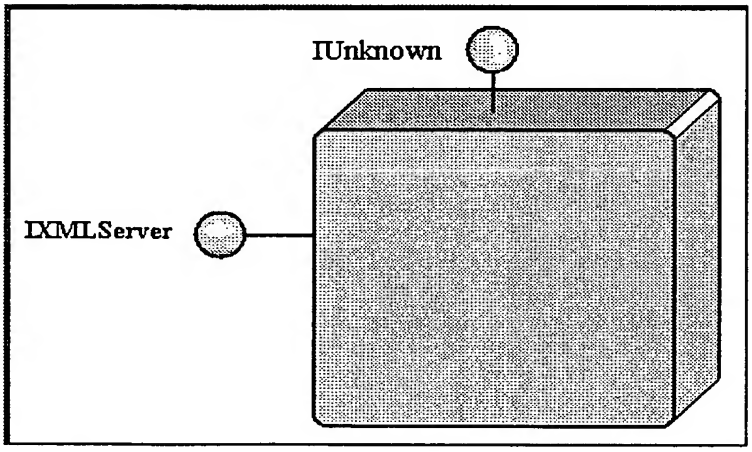


Abbildung 0.1.: Schnittstellen des XMLServers

(s) Der Export

(t) Der DOM-Parser

Der Export arbeitet mit dem Xerces Parser von Apache. Dieser Parser ist ein DOM- Parser, der sich dadurch auszeichnet, dass er einen XML- Baum aufbauen kann. Die Funktionalität des Parsers wird in verschiedenen Headerdateien beschreiben.

Durch das Einfügen dieser Dateien im Projekt stehen die Funktionen des DOM- Parsers zur Verfügung.
Die Klassen *DOM_Node*, *DOM_Document* und *DOM_Element* sind die wichtigsten Klassen des DOM- Parsers. Ein XML- File wird durch die Klasse *DOM_Document* repräsentiert. Ein Tag oder Element eines XML- Files wird durch ein *DOM_Element* realisiert, die in einem *DOM_Document* eingefügt werden können. Außerdem wird durch die Klassen *DOM_Attr* , *DOM_Entity*, *DOM_Text* und *DOM_Comment* weitere Funktionalität zur Verfügung gestellt, um einen Baum aufzubauen und ein XML- File zu erstellen. Fortführende Dokumentation zu dem DOM- Parser von Apache ist im Internet unter www.apache.de und www.w3.org/TR/1998/REC-DOM-Level-1-19981001/level-one-core.html zu finden.

Der Aufbau des XML- Baumes wird durch den Aufruf verschiedener Funktionen des DOM- Parsers erreicht. Deswegen beinhaltet die Klasse *CDomHelper* die Grundfunktionalität, um die XML-Files zu erstellen.
Ein wichtiger Member der Klasse ist die Instanz der Klasse *DOM_Document*, die eine XML- Datei erzeugt, und eine Instanz der Klasse *DOM_Element*, die das Roottag speichert.
Wie aus dem Code ersichtlich, sind verschiedene Funktionen zum Beschreiben eines XML- Files vorhanden, wie *writeDomDoc*, *writeDTD* und *writeXSL*. Sie sind private Funktionen der Basisklasse *CDomHelper* und werden am Ende des Exports einer Datenmenge aufgerufen. Somit wird eine XML- Datei erst nach dem vollständigen Export geschrieben, um keine inkonsistenten Daten in der Datei zu erhalten und damit die Fehlerbehandlung erleichtert wird. Das heißt, wenn ein Fehler beim Exportieren von Daten auftritt, können die fehlerhaften Daten verworfen werden, ohne das eine Datei von der Festplatte gelöscht werden muß.
Außerdem ist es möglich mit den Funktionen *openDomHelper* und *closeDomHelper* Vorinitialisierungen bzw. das abschließende Schreiben der XML- Datei durchzuführen. Die Funktion *openDomHelper* initialisiert die Membervariablen mit gültigen Werten

und legt ein DOM_Document mit einem DOM_Element an, das den Namen des übergebenen Strings besitzt. In der Methode *close-DomHelper* werden die oben genannten Funktionen zum Schreiben des XML- Files aufgerufen.

Um XLinks zu erstellen und neue Verzeichnisse anzulegen, stehen die Funktionen *createSubDir*, *setXLink* und *setFullXLink* zur Verfügung.

Die Funktionen *writeDTD* und *ManageNavigation* sind reine virtuelle Funktionen und müssen von allen Klassen, die von der Klasse *CDomHelper* abgeleitet sind, implementiert werden.

```
class CDomHelper
{
public:
        CDomHelper();
        virtual ~CDomHelper();

        DOM_Document *getDomDoc()           { return &m_domDoc;};
        DOM_Element     *getDomRootNode()   { return &m_domRootElement;};
        ofstream        *getOS()            { return &m_OS;};

        ////////////////////////////
        // Static Members
        static CExport *m_pExport;
        static bool m_bSwitchXSL;

protected:
        ofstream            m_OS;
        ofstream            m_XSL_OS;
        DOM_Document        m_domDoc;
        DOM_Element         m_domRootElement;
        bool                m_DTD_Check;

        virtual void        writeDTD()          = 0;
        virtual void        writeXSL();
        virtual HRESULT     ManageNavigation()  = 0;
```

Quellcode 0.1.: Klassendefinition der Klasse *CDomHelper*

Wenn man nun bestimmte Daten eines Projektes exportieren möchte, wird eine eigene Klasse von der Klasse *CDomHelper* sowie von der Klasse *CBasisEsHelper* abgeleitet. Die Klasse *CBasisEsHelper* ist wichtig, um an den Pointer auf die entsprechende Schnittstelle im Basis ES leichter zu gelangen, und wird im Kapitel (q) eingehend beschrieben.
Bei dem Export eines Projektes werden zwei verschiedene Arten von XML- Dateien erstellt. Es können Dateien als „Linkdateien" fungieren sowie als „Datendateien". Die „Linkdateien" beinhalten keinerlei Daten, sondern nur Links zu „Datendateien" und „Linkdateien", die in einem Unterverzeichnis liegen (siehe Kapitel 0).

Um eine „Datendatei" zu erstellen, wird die von den beiden oben genannten Klassen abgeleitete Klasse benötigt, die ein neues XML- Dokument anlegt und dies mit den zu exportierenden Daten füllt. Diese Daten sollten möglichst eine abgeschlossene Datenmenge bilden.
Diese Klasse muß die Funktion *ManageNavigation*, wie in den folgenden Abschnitten erläutert, überschreiben.

Die Funktion *ManageNavigation* ist der Einstieg in die Klasse. Hier soll dafür gesorgt werden, dass ein XML- Dokument angelegt wird, die Daten exportiert werden und das XML- File mit den exportierten Daten beschrieben wird.
Das Anlegen der XML- Datei wird, wie oben beschrieben, durch die Funktion *openDomHelper* angestoßen. Danach ist die Membervariable *m_domRootElement* auf einen gültigen Wert gesetzt, an die man nun weitere Elemente anfügen und somit den XML- Baum aufbauen kann.
Für das Exportieren einzelner Daten ist es möglich, eigene Funktionen zu erstellen und diese dann in der Methode *ManageNavigation* aufzurufen. Dabei kann man die Daten zum Einen direkt an das Rootelement, das wie gesagt in der Variable *m_domRootElement* gespeichert ist, anhängen oder sie anderen Falls unter ein neu erstelltes Element hängen, je nach Struktur der Daten im Basis ES. Das heißt, das die Funktionen den übergeordneten Knoten als Aufrufsparameter erhalten und sie sich ebenso tief wie die Elemente im XLM- Baum schachteln können.
Wenn alle Daten aus dem Basis ES gelesen wurden und keine Fehler auftraten, wird die Funktion *closeDomHelper* aufgerufen und somit die XML- Datei mit den Daten gefüllt.

Eine „Linkdatei" wird durch eine prinzipiell genauso aufgebaute Klasse erstellt. Die Funktion *ManageNavigation* ruft die Methoden *openDomHelper* und *closeDomHelper* auf, um die XML-Datei zu erstellen. Pro Link kann man eine Funktion anlegen, die in der Methode *ManageNavigation* aufgerufen wird. Diese Funktionen sollte einen XLink zu der entsprechenden Datei anlegen, eine Instanz der Klasse anlegen, die den Export der Daten durchführt, und deren *ManageNavigation* Funktion aufrufen.

Die DTD eines XML- Files wird mit der Funktion *writeDTD* erstellt, die wie schon erwähnt überschrieben werden muß. Für die DTD wird am Beginn des XML- Files ein String eingefügt,

in dem ein Makro an die dafür vorgesehene Membervariable angehängt wird. Dafür stehen die Dateien *DTDHelper.h* und *DTDObjectDefs.h* zur Verfügung. In der Datei *DTDHelper.h* sind verschiedene Makros definiert, die das Erstellen einer DTD vereinfachen sollen. In der *DTDObjectDefs.h* muß nur noch ein spezielles Makro für den DTD- String definiert werden, das dann in der Methode *writeDTD* verwendet wird.
Die XSL- Definitionen werden ähnlich erstellt und sind für die Daten wichtig, die in Internet- Explorer angezeigt werden sollen. Auch hier existieren zwei Haederdateien *XSLDefs.h* und *XSLDefsHelper.h*, die Strings für die XSL- Files definieren. In der *XSLDefsHelper.h* sind Makros definiert, die das Erstellen der XSL- Datei erleichtern. Die Makros für die jeweilige XML- Datei werden in der *XSLDefs.h* Datei eingefügt. Es muß darauf geachtet werden, dass die neue XSL- Definition an entsprechender Stelle in eine andere eingefügt wird, wenn für die XML- Datei kein eigener Ordner angelegt wird und sie sich mit anderen XML- Files eine XSL- Datei teilen muß. Eine eigene XSL- Datei wird nur von den Klassen, die eine „Linkdatei" erstellen, durch die Funktion *writeXSL* angelegt.

(u) Navigation über ein Projekt

Für die Beschaffung der Pointer auf die Schnittstelle, die für den Export angesprochen werden muß, stehen den Klassen der Export- Import- Software die Klasse *CBasisEsHelper* zur Verfügung.

```
class CBasisEsHelper
{
public:

// Konstruct / Destruct
        CBasisEsHelper();
        virtual ~CBasisEsHelper();

//static Basemembers
public:

        static CComPtr<IMC_Project>   m_pProject;
        static CComPtr<IMC_Device>    m_pDevice;
        static CComPtr<IMC_Ident>     m_pTO;
        static CComPtr<IMC_Program>   m_pProgram;

        static CXMLServer* m_pXMLServer;

        static CComPtr<IUMCWBObjectSetHandler> m_pObjectSetHandler;
        static long m_lObjectSetHandle;              // Handle auf das Objectset

        static CComPtr<IUMCSnapInHost>       m_pSIH;
        static CComPtr<IUMCSnapIn>           m_pSI;

        static UMC_PROJECTMODE m_ProjectMode;

        static std::map<_bstr_t,_bstr_t>     m_bstrChangedDeviceNameList;

        static IsNameMapEntry*m_pTOMap;
        static long                          m_nTOMapSize;

        static unsigned long                 m_ExportFilter;
```

Quellcode 0.2.: Klassendefinition der Klasse *CBasisEsHelper*

Diese Klasse besitzt einige Variablen, die auf wichtige Schnittstellen zeigen. Von ihnen aus können andere Interfaces angesprochen werden. Darüber sind die benötigten Schnittstellen zu finden. Die wichtigsten Interfacepointer sind zu den Schnittstellen *IMC_Project*, *IMC_Device*, *IMC_Ident* und *IMC_Program*. Außerdem gibt es noch Pointer zum *Snap- In*, zum *Snap- In- Host* und zum *XMLServer* für die Ausgaben beim Im- und Export. Um die Daten bearbeiten zu können, müssen sie an ein sogenanntes „ObjectSet" der Schnittstelle *IUMCWBObject-SetHandler* gebunden werden. Auf dieses Interface zeigt die Membervariable *m_pObjectSetHandler*.

Neben den Variablen sind verschiedene Funktionen definiert. Daten, die in ein XML- File geschrieben werden, müssen in Strings vorliegen. Deswegen sind in der Klasse *CBasisEsHelper* mehrere Methoden zum Konvertieren von Daten in Strings und umgekehrt vorhanden. Für das „ObjectSet" sind verschiedene Funktionen definiert, die Objekte dem „ObjectSet" hinzufügen und entfernen können.

Die Daten eines Projektes sind unter verschiedenen Schnittstellen abgelegt. Um an ein Interface zu gelangen, das die gewünschten Daten enthält, ist es manchmal notwendig über verschiedene andere Schnittstellen zu navigieren. Für das im Kapitel (c) beschriebene Beispiel, war es nötig über zwei weitere Schnittstellen das Interface anzufordern. Wie aus der Abbildung 0.2 ersichtlich, ist es möglich die Schnittstellen durch Aufrufen von *QueryInterface* und *getObjectByID* zu erhalten.

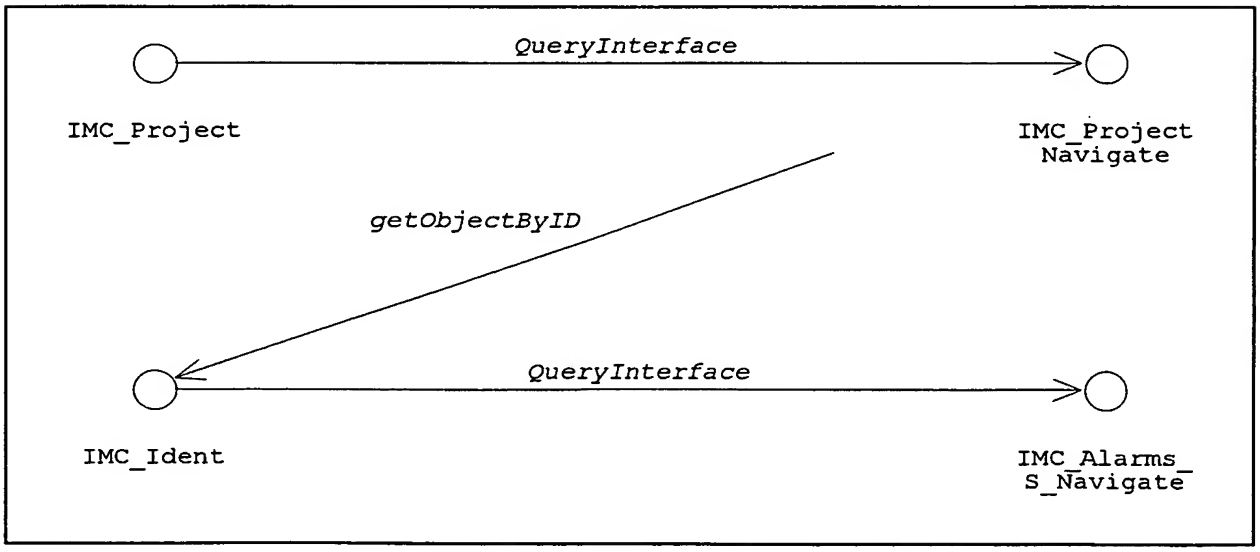


Abbildung 0.2.: Navigieren zu den Interface für den Export der Alarme

Eine Dokumentation über die Schnittstellen des Basis ES und eine Beschreibung, wie verschiedene Interfaces angesprochen

werden, ist unter
\\erlf452a\umc all$\doc\ICD\icd basis es.doc vorhanden.

(v) Ein Beispiel für den Export – exportieren von Alarmen

Um die Funktionsweise und den Aufbau des Exports verständlicher zu machen, erläutere ich den Export am Beispiel des Exports von Alarmdaten.
Die Stelle, an der das Programm erweitert werden muß, ist abhängig von den Daten, die exportiert werden sollen, da der Export die Dateien in einer Verzeichnishierarchie ablegt, die der Hierarchie des Projektes im SIMOTION SCOUT- Programm ähnlich ist (siehe Kapitel 0). Die Alarme sind projektglobal, deswegen sind sie auf der ersten Ebene zu exportieren.

Da die Alarme nicht von anderen Daten im Projekt abhängig sind, legt der Export für diese Daten eine neue XML- Datei an. Deswegen ist es vorteilhaft für den Export der Alarme eine neue Klasse anzulegen. Diese Klasse heißt *CAlarmNavigation*. Sie besitzt alle Eigenschaften, die eine „Navigation-Klasse" hat und ist wie folgt aufgebaut.

```
class CAlarmNavigation :
        public CBasisEsHelper,
        public CDomHelper
{
public:
        //=========================================================================
        //Construction/Destruction
        //=========================================================================
        CAlarmNavigation(_bstr_t bstrExportPath);
        virtual ~CAlarmNavigation();


        //=========================================================================
        // public methods
        //=========================================================================
        HRESULT ManageNavigation();


private:
        //=========================================================================
        // private methods
        //=========================================================================
        HRESULT getTargetLanguage();
        HRESULT getCurrentLanguage();
```

Quellcode 0.3.: Klassendefinition der Klasse *CAlarmNavigation*

Der Einstiegspunkt für den Export eines Projektes ist die Klasse *CProjectNavigation*. Projektglobale Daten werden, von dieser Klasse ausgehend, exportiert. Das heißt die Klasse *CAlarmNavigation* wird in der Klasse *CProjectNavigation* angelegt und abgearbeitet. Dafür wird eine neue Funktion *getAlarms* in der Klasse *CProjectNavigation* angelegt. Die Funktion *ManageNavigation* der Klasse *CProjectNavigation* muß um den Aufruf der Funktion *getAlarms* erweitert werden.

```
if ((m_ExportFilter & XMLDATA_PROJECTEXT) == XMLDATA_PROJECTEXT) {

        hr = getAlarms();
```

Quellcode 0.4.: Funktion *ManageNavigation* in der Klasse *CProjektNavigation*

Die Funktion *getAlarms* der Klasse *CProjectNavigation* erstellt einen Link zu der XML- Datei der Alarme. Er zeigt auf die XML- Datei, die später die Alarme enthält. Über diesen Link ist es möglich vom Projekt auf die Alarme zuzugreifen, was für den Import wichtig ist. Außerdem muß die Funktion *getAlarms* eine Instanz der Klasse *CAlarmNavigation* anlegen und dessen *ManageNavigation* Funktion aufrufen.

```
HRESULT CProjectNavigation::getAlarms()
{
        HRESULT hr = S_OK;

        // Create Dom Node ProjecData
        DOM_Element domAlarms = m_domDoc.createElement("UMC_Alarms");
        m_domRootElement.appendChild(domAlarms);

        setXLink(domAlarms,_T("Alarms"),XLINK_FILE);
```

Quellcode 0.5.: Funktion *getAlarms* der Klasse *CProjektNavigation*

Im Konstruktor der Klasse *CAlarmNavigation* wird eine neue XML- Datei angelegt, die Alarms.xml heißt. Außerdem wird die Membervariable *m_bstrDataExportPath* gesetzt, die den Pfad der Datei angibt.

Nach dem Durchlaufen des Konstruktors durch das Anlegen einer Instanz der Klasse *CAlarmNavigation* wird nun die Funktion *ManageNavigation* aufgerufen.

Diese Funktion übernimmt die Organisation des Exports. Die Daten, der Alarme liegen im Basis ES. Es muß also zuerst ein Pointer auf das Interface angefordert werden. Danach werden alle relevanten Daten mit Hilfe einzelner Funktionen aus dem Basis ES gelesen.

```
HRESULT CAlarmNavigation::ManageNavigation()
{
        HRESULT hr = S_OK;

        hr = getInterfacePointer();

        if(FAILED(hr))
        {
                m_pExport->throw_Error(get_ResString(IDS_EX_ERR_NAVALARMS));
                return E_FAIL;
        }

        hr = openDomHelper(_T("Alarms")); // root Node Name

        hr = getVersion();
        if (FAILED(hr))
                return E_FAIL;


        hr = getCurrentLanguage();
        if (FAILED(hr))
```

Quellcode 0.6.: Funktion *ManageNavigation* der Klasse *CAlarmNavigation*

Den Schnittstellenzeiger auf das Interface *IMC_Alarm_S_Navigate* erhält man nur über Umwege. Ausgehend vom Projekt wird ein *QueryInterface* auf die Schnittstelle des Projektnavigators *IMC_ProjectNavigate* gemacht. Der Zeiger auf diese Schnittstelle enthält die globalen Objekte, die mit dem Aufruf der Funktion *getObjectByID* angesprochen wird. Hier ist es nun möglich ein *QueryInterface* auf das gewünschte Interface zu machen und dies einer Membervariable zuzuweisen.

```
HRESULT CAlarmNavigation::getInterfacePointer()
{
        // get a pointer to the interface of projectnavigation
        CComQIPtr<IMC_ProjectNavigate,& IID_IMC_ProjectNavigate> pPN(m_pProject);

        if(pPN == 0)
                return E_FAIL;

        IMC_Ident * pIdent = NULL;
        DWORD ret = 0;

        // get a pointer to the interface of the global objects
        // form the projectnavigation
        HRESULT hr = pPN->getObjectByID(ESObjectID_PmGlobCommon,&pIdent,&ret);

        if((FAILED(hr)) || (ret != BUMC_OK))
        {
                SafeRelease(pIdent);  // freigeben
                return E_FAIL;
```

Quellcode 0.7.: Schnittstellenzeiger für die Alarme

Wenn der Schnittstellenzeiger initialisiert ist, können die Daten aus dem Basis ES gelesen werden. Für den Export der Alarme sind verschiedene Daten wichtig, wie die *Version*, die *CurrentLanguage*, die *TargetLanguage*, die *NumberRange* und die Alarmdaten an sich.

Die Alarme gliedern sich in verschiedene Sprachen. Das heißt zuerst muß man sich von der Schnittstelle die Liste der Alarme (Funktion *getAlarmsList*) holen. Dann ist es möglich mit Hilfe der Daten eines Alarms (Funktion *getAlarm*) die verschiedenen Sprachen eines Alarms (Funktion *getLanguageList*) zu erhalten. Ausgehend von dem Alarm kann man jetzt den Text in der zugehörigen Sprache vom Interface erfragen (Funktion *getAlarmText*). Jede Funktion sollte die Daten, die sie erhalten hat, in das XML- File schreiben.

Der oberste Knoten heißt Alarm und wird in der Funktion *getAlarmsList* gefüllt.

```
HRESULT CAlarmNavigation::getAlarmsList()
{
        HRESULT hr              = S_OK;
        long size               = 0;
        ESAlarmID * IDlist      = NULL;

        // get the list of the alarmIDs
        hr = m_pIAlarm->GetAlarmIDList(&size,&IDlist);

        if((FAILED(hr)) || (IDlist == NULL))
        {
                m_pExport->
                        throw_Error(get_ResString(IDS_EX_ERR_ALARM_LIST));
                return E_FAIL;
        }

        // get the Alarm for each ID in the list
        for(int i = 0; i < size; i++)
        {
                char string[256] = {0};

                // create a new element
```

Quellcode 0.8.: Funktion *getAlarmsList* der Klasse *CAlarmNavigation*

Wie man in diesem Ausschnitt sehen kann, wird das *DOM_Element* der *getAlarm* Funktion mit gegeben. Diese Funktion kann somit ein *DOM_Element* unter das übergebene hängen. Außerdem ruft die Funktion *getAlarm* die Funktion *getLanguageList* auf und übergibt dieser das neue *DOM_Element*.
Die Funktion *getLanguageList* hängt wiederum ein neues *DOM_Element* an das übergebene. Somit wird DOM- Baum aufgebaut, dessen letztes Element der Alarmtext in einer bestimmten Sprache ist.

```
HRESULT CAlarmNavigation::getAlarmText(DOM_Element * parentNode, ESAlarmLanguageID Langua-
geID, ESAlarmID dwAlarmID)
{
        HRESULT hr                      = S_OK;
        DWORD ret                       = 0;
        ComMCAlarmSTextData TextData;

        // get the text of the alarm
        hr = m_pIAlarm->GetRawText(dwAlarmID,LanguageID,&TextData,&ret);

        // connection to the interface was succseeded
        if(SUCCEEDED(hr))
        {
                // can´t found the text of the alarm
                if(ret == BUMC_NOT_FOUND)
                {
                        wchar_t buffer[256]    = {0};
                        char alarmstring[256] = {0};
                        char langstring[256]  = {0};

                        _ultoa(dwAlarmID,alarmstring,10);

                        if(GetLocaleInfo(MAKELCID(LanguageID,SORT_DEFAULT),
                                                LOCALE_SLANGUAGE,langstring,255) == 0 )
                                strcpy(langstring,"");

                        _snwprintf(buffer,sizeof(buffer),
                                        get_ResString(IDS_EX_WAR_ALARM_TEXT),
                                        m_szAlarmName,langstring);

                        m_pExport->throw_Error(buffer);

                        return E_FAIL;
```

Quellcode 0.9.: Funktion *getAlarmText* der Klasse *CAlarmNavigation*

Mit diesen Schritten ist es möglich alle Daten der Alarme eines Projektes zu exportieren. Um die Datei im Internet Explorer anzuzeigen, muß jedoch eine DTD und ein XSL- File vorhanden sein. Für die DTD existiert die Funktion *writeDTD* in der Klasse *CDomHelper*, die überschrieben werden muß. Um den String der DTD zusammen zusetzten, kann man in der Datei *DTDObjectDefs.h* eine #define anfügen. Die grundlegenden Defines sind in der Datei *DTDHelper.h* schon definiert.

```
////////////////////////////////////////////////////////////////////////////
//      Alarms
//
#define DTD_ALARMS                                                          \
                                                                            \
                DTD_D("Alarms")                                             \
        <<      DTD_E("Alarms (Version,CurrentLanguage,TargetLanguage,IDRange?,"    \
        <<              "Alarm*)")                                          \
        <<      DTD_E("Version (#PCDATA)")                                  \
        <<      DTD_A("Version GUID CDATA #REQUIRED Major CDATA #IMPLIED "  \
        <<              "Minor CDATA #IMPLIED")                             \
        <<      DTD_E("CurrentLanguage (#PCDATA)")                          \
        <<      DTD_A("CurrentLanguage LanguageID CDATA #REQUIRED "         \
        <<              "Name CDATA #REQUIRED")                             \
        <<      DTD_E("TargetLanguage (#PCDATA)")                           \
        <<      DTD_A("TargetLanguage LanguageID CDATA #REQUIRED "          \
```

Quellcode 0.10.: DTD der Alarme

Das XSL- File wird auch durch Defines erstellt. Diese Defines stehen in der Datei *XSLDefs.h* und basieren auf Defines der *XSLDefsHelper.h* Datei.

```
////////////////////////////////////////////////////////////////////////////////
// ALARMS XSL
//
#define XSL_ALARMS                                                              \
                                                                                \
              XSL_ALARMS_LANGUAGE                                               \
       <<     XSL_T("Alarms",                                                   \
              XSL_HTML_FONT ("Arial Black","3","Alarms")                        \
       <<     "    <TABLE STYLE=\"border:1px solid black\">\n"                  \
       <<     "        <TR STYLE=\"font-size:12pt; font-family:Verdana; "      \
       <<     "font-weight:bold\">\n"                                          \
       <<     "            <TD>Symbol</TD>\n"                                  \
       <<     "            <TD STYLE=\"background-color:lightgrey\">Language</TD>\n"  \
       <<     "        </TR>\n"                                                \
       <<     "        <xsl:for-each select=\"Alarm\">\n"                      \
       <<     "        <xsl:for-each select=\"Alarmdata\">\n"                  \
       <<     "            <TR>\n"                                             \
       <<     "                <TD><xsl:value-of select=\"@Symbolname\"/></TD>\n"  \
       <<     "                <TD STYLE=\"background-color:lightgrey\">\n"    \
```

Quellcode 0.11.: XSL- Datei der Alarme

(w)Der Import

(x) Der SAX- Parser

Der SAX- Parser ist ein ereignisgesteuerter Parser, der beim Auftreffen auf ein XML- Tag in eine Funktion verzweigt. Es wird kein XML- Baum im Speicher gehalten, was der Performance zu Gute kommt. Durch diese Funktionalität ist dieser Parser für den Import eines Projektes geeignet.
Um den SAX- Parsers zu nutzen, sind die Headerdateien des Parsers im Projekt eingezogen. Der Parser besteht aus der Klasse *HandlerBase* und besitzt mehrere Funktionen, die von ihm bei bestimmten Ereignissen aufgerufen werden. Die Funktionen sind nur definiert und die endgültige Implementierung wird erst von dem Programmierer in einer abgeleiteten Klasse

durchgeführt. Damit kann man die Funktionen speziell auf das zu lösende Problem zuschneiden. Weiterführende Dokumentation dazu ist auf der Seite www.apache.de oder www.megginson.com/SAX/index.html zu finden.

In der XML- Software ist die Klasse *CSAXHandler* von der Klasse *HandlerBase* abgeleitet. Diese Klasse überschreibt, wie aus dem Code ersichtlich, alle Funktionen der Klasse *HandlerBase*. Wichtig für die Software sind davon allerdings nur die Funktionen *startElement*, *endElement*, *characters* und für die Fehlerausgabe die Funktionen *warning*, *error* und *FatalError*.

```
class CSAXHandler : public HandlerBase, public CImportBase
{
public:
        bool ElementIsNormalTOType(_bstr_t bstrElementName);
        bool ElementIsNormalTO(_bstr_t bstrElementName);
        HRESULT GetResult();
        void SetParseFile(_bstr_t bstrFile);
        bool GetDoValidation() { return m_bDoValidation; }

//Attributes
        static CParseManager * m_pParseManager;
        static IsNameMapEntry * m_pTONameMap;
        static long m_lTONameMapSize;

        CSAXHandler();
        CSAXHandler(_bstr_t bstrHandlerType);
        CSAXHandler(_bstr_t bstrHandlerType, bool bDoValidation);
        virtual ~CSAXHandler();

protected:
        _bstr_t m_bstrParseFile;
        bool m_bErrorOccured;
        _bstr_t m_bstrHandlerType;
        bool m_bDoValidation;

public:
        // ------------------------------------------------------------------------
   //  methods to be overwridden by the derived classes
   // ------------------------------------------------------------------------
        virtual void HandleStartElement(const XMLCh* const name, AttributeList& attributes);
        virtual void HandleEndElement(const XMLCh* const name);
        virtual void HandleCharacters(const XMLCh* const chars, const unsigned int length);
```

Quellcode 0.12.: Klassendefinition der Klasse *CSAXHandler*

Der Import beginnt in der Klasse *CParseManager* durch den Aufruf der Funktion *ParseProject*. Hier werden einige Initialisierungen vorgenommen und der Import des gesamten Projektes wird angestoßen.

Beim Parsen eines Startelementes ruft er SAX- Parser die Funktion *startElement* der Klasse *CSAXHandler* auf. Die Funktion unterscheidet, ob das Tag ein Link oder ein normales Element ist.

Wenn ein Link gefunden wird, stößt die Funktion das Parsen der im Link enthaltenen Datei an. Um diese Funktionalität bei allen Dateien mit Links zu gewährleisten, wird für jede XML-Datei eine sogenannte „Handlerklasse" erstellt, die von der Klasse *CSAXHandler* abgeleitet ist. Ist das gefundene Tag ein Link, wird ein Pointer auf die Handlerklasse initialisiert, die diese XML- Datei parsen soll. Danach wird das Parsen der Datei durch die Funktion *ParseXLinks* der Klasse *CParseManager* angestoßen.
Beim Parsen von Elementen muß man jedoch anders vorgehen. Dafür stehen die Funktionen *HandleStartElement*, *HandleEndElement* und *HandleCharacters* zur Verfügung. Sie werden von den „Handlerklassen" entsprechend überschreiben. Wenn die Funktionen *startElement*, *endElement* und *characters* mit einem Element aufgerufen wird, rufen sie die entsprechende der drei Funktionen *HandleStartElement*, *HandleEndElement* und *HandleCharacters* auf.

(y) Handeln von Projektkomponenten

Zum Importieren sollte eine „Handlerklasse" pro zu importierender XML- Datei existieren. Diese Klassen sind von der Klasse *CSAXHandler* abgeleitet und erben somit die Funktionalität der Basisklasse.
Wie beim Export kommt es beim Import auch auf den Inhalt der Dateien an. Bei einer Datei, die ausschließlich Links enthält, wird die Funktion *startElement* der Basisklasse um IF-Abfragen erweitert. Wenn dagegen XML- Files mit Daten geparst werden sollen, kann man die Klassen wie folgt anlegen.
Die Klassen müssen die Funktionen *HandleStartElement*, *HandleEndElement* und wenn nötig *HandleCharacters* überschreiben. Sie werden aufgerufen, wenn der Parser auf ein Start- oder Endelement bzw. auf einen Text getroffen ist. Dann ist es möglich in einer dieser Funktionen abzufragen, welches Tag der Parser gefunden hat. Somit kann dort eine Funktion zum

Auslesen der Daten aus dem gefundenen Element aufgerufen werden. Um das Auslesen zu kapseln, sollten die Funktionen in einer sogenannten „Makeklasse" definiert werden. Die „Handlerklasse" besitzt als Membervariable einen Pointer auf die „Makeklasse", die genau die Daten importiert, die das XML-File beinhaltet. Der Pointer wird im Konstruktor der „Handlerklasse" angelegt und im Destruktor freigegeben. Wenn ein Element gefunden wurde, das Daten für den Import enthält, dann wird in einer der drei oben genannten Funktionen eine Funktion der „Makeklasse" aufgerufen.

```
class CAlarmHandler : public CSAXHandler
{
public:
        //=========================================================================
        //Construction/Destruction
        //=========================================================================
        CAlarmHandler();
        virtual ~CAlarmHandler();

        //=========================================================================
        // public methods
```

Quellcode 0.13.: Beispiel für eine „Handlerklasse"

(z) Anlegen von Projektkomponenten mit „Makeklassen"

Um die Daten zu importieren werden die Funktionen einer sogenannten „Makeklasse" aufgerufen. Ein Beispiel für eine solche „Makeklasse" ist im Quellcode 0.17 vorhanden.
Diese Klassen sind von der Klasse *CBasisEsHelper* abgeleitet. Damit sind alle Member und Funktionen der *CBasisEsHelper* Klasse der „Makeklasse" bekannt. Dies ist notwendig da die Daten in das Basis ES geschrieben werden müssen. Die Navigation zu dem Pointer auf die Schnittstelle, die beim Schreiben der Daten angesprochen wird, ist im Kapitel (b) schon für den Export erklärt. Außerdem befinden sich die Schnittstellen zum

Schreiben und Lesen meist an dem gleichen Interface. Deswegen ist die Dokumentation über die

Navigation zu den gewünschten Interface nicht nochmals in diesem Kapitel enthalten. Der Pointer kann nun in einer Membervariable gespeichert werden, da eine Instanz der „Makeklasse" während des Parsens eines XML- Files im Speicher bestehen bleibt.
Die „Makeklassen" benötigen nur Funktionen, die Daten eines Tags importieren. Es ist möglich Daten beim Auftreffen auf ein Element in das Basis ES zu schreiben oder beim Finden eines Endtags. Dies ist wichtig, wenn Daten von Elementen, die darunter geschachtelt sind, zum Anlegen der Daten im Basis ES benötigt werden.
Somit kann man für jedes Starttag und Endtag eine eigene Funktion definiert werden. Darin werden die Daten der Elemente ausgelesen. Dabei gibt es mehrere Möglichkeiten, wie die Daten im Tag eingebunden sind. Wenn sich ein Element aus anderen zusammensetzt, dann sind die Daten in den darunterliegenden enthalten. Jedes Tag kann natürlich auch eigene Daten besitzen, die in dessen Attributen gespeichert sind. Die einfachste und herkömmlichste Art die Daten eines Elements zu speichern, ist jedoch im Text zwischen Start- und Endtag. Wenn die Daten aus dem Tag gelesen wurden, können sie mit Hilfe des Pointers auf die entsprechende Schnittstelle in das Basis ES geschrieben werden. Wenn dabei Fehler auftreten, sollte auch hier eine Fehlerausgabe erfolgen.

(aa) Ein Beispiel für den Import – importieren von Alarmen

Wie beim Export soll auch an dieser Stelle ein Beispiel das in diesem Kapitel Beschriebene verdeutlichen.
Der Import ist ereignisgesteuert und ruft bei einem Ereignis (Element) die Funktion *startElement* der Klasse *CSAXHandler* auf. Hier werden die Links ausgewertet. In diesem Beispiel ist das Ereignis *„Link auf eine Datei Alarms.xml"* wichtig, der das Auslesen dieser Datei anstößt.

```
else if (bstrElementName == _bstr_t(U7DXML_ALARMS))
```

Quellcode 0.14.: Importieren des Links auf die Datei Alarms.xml

Die Variable *pEventHandler* der *CSAXHandler* Klasse wird mit einem Pointer auf die jeweilige „Handlerklasse" initialisiert. In dem Fall der Alarme ist dies die Klasse *CAlarmHandler*. Wenn dieser Pointer initialisiert wurde, kann die Funktion *ParseXlink* der Klasse *CParseManager* aufgerufen werden, die dafür sorgt, dass das im Link enthaltene File geparst wird.

```
if (pEventHandler)
{
        //  parsing Xlink file with the created parse handler
```

Quellcode 0.15.: Starten des Parsens eines XML- File

Wenn die Funktion *startElement* der Basisklasse *CSAXHandler* jetzt aufgerufen wird, verzweigt diese die Funktion *HandleStartElements*. Ebenso wird bei *endElement* der *CSAXHandler* die Methode *HandleEndElements* aufgerufen. Die Klasse *CAlarmHandler* überschreibt diese Funktionen, so dass dort eine spezielle Bearbeitung der XML- Datei umgesetzt werden kann. Durch den Pointer auf die „Makeklasse" *CMakeAlarm* kann auf Funktionen zugegriffen werden, die Alarme im Basis ES anlegen.

```
void CAlarmHandler::HandleStartElement(const XMLCh *const name, AttributeList &attributes)
{
        _bstr_t bstrElementName(name);

        if (bstrElementName == _bstr_t(U7DXML_CURRENT_LANGUAGE))
        {
                UMCINFO "HandleStartElement for: %s", (char*)bstrElementName END
                m_pMakeAlarm->setCurrentLanguage(name, attributes);
        }
        else if (bstrElementName == _bstr_t(U7DXML_TRAGET_LANGUAGE))
        {
                UMCINFO "HandleStartElement for: %s", (char*)bstrElementName END
                m_pMakeAlarm->setTargetLanguage(name, attributes);
        }
        else if (bstrElementName == _bstr_t(U7DXML_IDRANGE))
        {
                UMCINFO "HandleStartElement for: %s", (char*)bstrElementName END
                m_pMakeAlarm->setIDRange(name, attributes);
        }
        else if (bstrElementName == _bstr_t(U7DXML_ALARM))
```

Quellcode 0.16.: Auslesen der Anfangselemente

Den eigentlichen Import der Projektdaten übernimmt die Klasse *CMakeAlarm*. Wenn ein Element im XML- File gefunden wurde, das Daten enthält, die für den Import wichtig sind, wird eine Funktion der Klasse *CMakeAlarm* aufgerufen, die diese Daten ins Basis ES schreibt.

```
class CMakeAlarm : public CBasisEsHelper
{

public:
        //=========================================================================
        //Construction/Destruction
        //=========================================================================
        CMakeAlarm();
        virtual ~CMakeAlarm();

        //=========================================================================
        // public methods
        //=========================================================================
        void setAlarmText(const XMLCh *const name, AttributeList &attributes);
        void endAlarmText(const XMLCh *const name);

        void setLanguage(const XMLCh *const name, AttributeList &attributes);
        void endLanguage(const XMLCh *const name);

        void setAlarmData(const XMLCh *const name, AttributeList &attributes);
        void endAlarmData(const XMLCh *const name);
```

Quellcode 0.17.: Importfunktionen für Alarme

In den Methoden der Klasse *CMakeAlarm* werden die Daten aus den XML- Elementen gelesen und über ein Interface in das Basis ES geschrieben. Die Schnittstelle, die das Schreiben der Daten ermöglicht, heißt *IMC_Alarm_S_Edit* und ist über die selben Schnittstellen zu erreichen, wie die beim Export benötigte Schnittstelle. An diesem Interface werden dann die Funktionen zum Schreiben der Alarmdaten aufgerufen.

```
void CMakeAlarm::setAlarmData(const XMLCh *const name, AttributeList &attributes)
{
        HRESULT hr = S_OK;
        DWORD ret = 0;

        ComMCAlarmSData alarmdata;

        // get attributes
        _bstr_t bstrSymbolName = attributes.getValue("Symbolname");
        alarmdata.SymbolName = bstrSymbolName;

        _bstr_t bstrMsgType = attributes.getValue("MsgType");
        alarmdata.MsgType = atoi((char*)bstrMsgType);

        _bstr_t bstrMsgClass = attributes.getValue("MsgClass");
        alarmdata.MsgClass = atoi((char*)bstrMsgClass);

        _bstr_t bstrMsgText = attributes.getValue("MsgTextLocked");
        alarmdata.Msgtextlocked = atoi((char*)bstrMsgText);

        _bstr_t bstrInfoText = attributes.getValue("InfoTextLocked");
```

Quellcode 0.18.: Schreiben von Daten in das Basis ES

Kommunikation mit Step7

Die dritte und letzte Dll der Export- Import- Software, ist für den Export und Import von den bei SIMOTION SCOUT enthaltenen Step7- Anteilen zuständig. Die Hardwarekonfiguration eines Projektes kann mit der Exportfunktion von Step7 exportiert werden. Für den Import steht ebenso eine Funktion von Step7 zur Verfügung. Die Daten werden im ASCII- Format gespeichert. Deswegen können die schon vorhandenen Funktionen von Step7 für den XML- Export- Import verwendet werden.
Die Schnittstelle, die dafür von Außen angesprochen werden muß, ist IXMLS7Project. Sie stellt die zwei Funktionen zum Export bzw. zum Import der Daten zur Verfügung. Deswegen heißen diese Funktionen *Import* und *Export*.

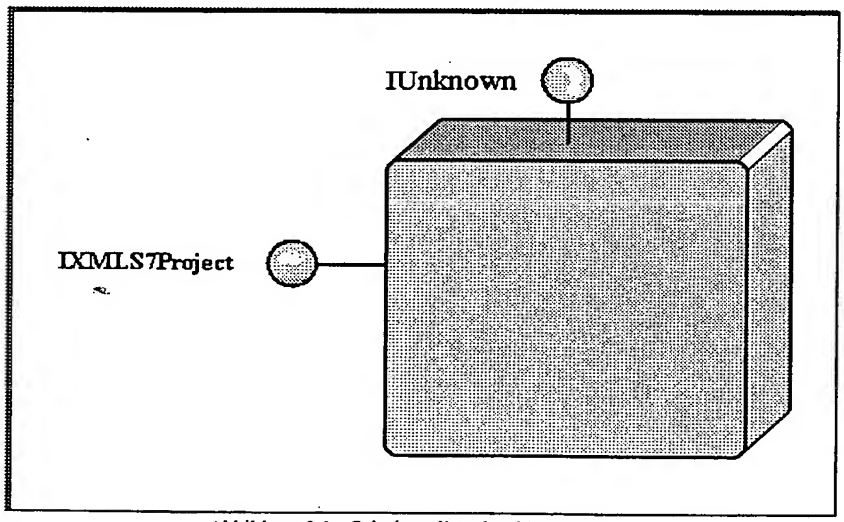


Abbildung 0.1.: Schnittstellen des S7Servers

Um bei dem Export und dem Import nicht auf direkt auf das Step7 zuzugreifen zu müssen, übernimmt ein sogenannter *Daemon* diese Aufgaben. Es werden Funktionen des *Daemon* aufgerufen, dieser „übersetzt" die Aufrufe und führt die entsprechenden Aktionen bei Step7 durch.

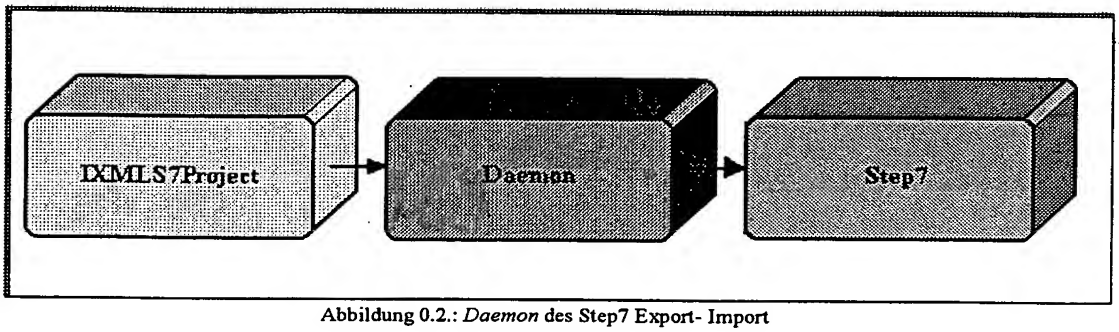


Abbildung 0.2.: *Daemon* des Step7 Export- Import

Der Export sowie der Import der Step7- Projekte werden in der Klasse *CXMLS7Project* durchgeführt.

Für den Export wird in dieser Klasse die Funktion *Export* aufgerufen. Hier werden die Funktionen *OpenProjectS7*, *ChooseRelevantStations*, *ExportRelevantStations* und *CloseProjektsS7* aufgerufen. Damit wird zuerst das zu exportierende Projekt und die darin enthaltenen Stationen ausgewählt. Dann werden die Stationen durch *ExportRelevantStations* exportiert. Das Schließen des Projektes wird mit Hilfe der Methode *CloseProjektsS7* ermöglicht.

Beim Importieren wird die Funktion *Import* aufgerufen. Das Öffnen und Schließen des Step7- Projektes übernehmen auch hier die Funktionen *OpenProjectS7* und *CloseProjektsS7*.Weiterhin werden die Funktionen *GetCurrentS7Handle*, *CompareStationNames* und *ImportOneStation* verwendet. Die Methode *CompareStationNames* wird benötigt, um das Importieren einer schon im Projekt vorhandenen Station zu verhindern. Wenn Stationen importiert werden sollen, muß die Methode *ImportOneStation* für die zu importierende Station aufgerufen werden.

Das Format und die Hierarchie der exportierten Daten

Ein exportiertes SIMOTION SCOUT- Projekt wird in einer Ordnerstruktur abgelegt. Diese Struktur ist angelehnt an den Aufbau eines Projektes im SIMOTION SCOUT- Projektnavigator.

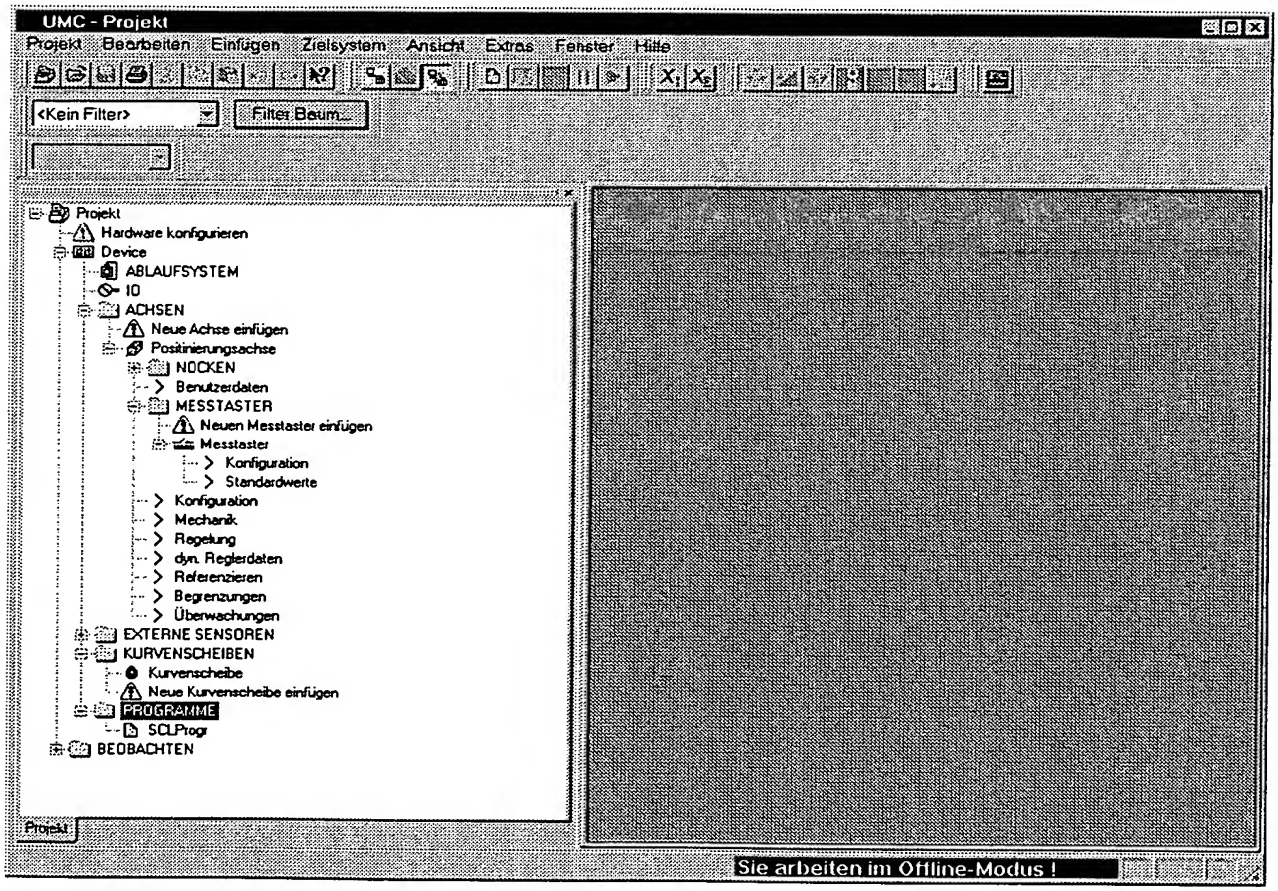


Abbildung 0.1.: Aufbau eines Projektes im Projektnavigator

Die große Menge der Daten eines Projektes wird in gegliederten Datenblöcken gespeichert. Damit wird eine übersichtliche Datenhaltung möglich gemacht, die dem Benutzer nach dem Export immer noch logisch erscheint. Ein anderer Gesichtspunkt für eine solche Hierarchie, ist die Möglichkeit, Komponenten eines Projektes zu analysieren oder über das Internet zu verschicken, ohne das unwichtige Daten stören.

(bb) Die Hierarchie der exportierten Daten eines SIMOTION SCOUT- Projektes

Ordner: MyProjectDataFolder
- Projekt.xml
- Ordner: Projekt
  - Projekt.xml
  - Alarms.xml
  - Installation.xml
  - ReferenceTable.xml
  - S7ProjectData.xml
  - Ordner: Device
    - Device.xml
    - RunTimeLevels.xml
    - DeviceData.xml
    - Device_Symbols.xml
    - Ordner: ProcessingUnit
      - ProcessingUnit.xml
      - SCL-Programm.xml
    - Ordner: TOCamType
      - TOCamType.xml
      - Kurvenscheibe.xml
    - Ordner: TOPosAxis
      - TOPosAxis.xml
      - Ordner: Positionierungsachse
        - Positionierungsachse.xml
        - ExpertList.xml

(cc) Erläuterung zu den exportierten Dateien

In der Hierarchie werden unterschiedliche XML- Dateien angelegt. Dateien können ausschließlich Links enthalten oder ausschließlich Daten. Die Dateien, die Links enthalten, liegen in einem gleichnamigen Ordner und sind die Dateien, durch die der Einstieg in diesen Ordner ermöglicht wird. Die Links, die in den Dateien enthalten sind, verweisen auf XML- Dateien, die für die Datenhaltung zuständig sind und im gleichen Ord-

ner liegen. Außerdem können die Links auch auf die Einstiegsdateien der Unterordner zeigen. Durch dieses Prinzip wird die Portierbarkeit vereinfacht. Das heißt es ist gewährleistet, einen Ordner aus der Hierarchie zu nehmen, ohne das die Struktur verloren geht. Somit können relevante Daten an Dritte weiter gegeben werden ohne unnütze Dateien mit zuliefern.

Ordner: MyProjectDataFolder - Der Ordner enthält das gesamte exportierte Projekt mit allen XML-Dokumenten, XSL- Dateien, HTML- Seiten und gegebenenfalls Unterordnern, die alle Daten des Projektes mit dessen speziellen Eigenschaften im XML- Format beschreiben. Der Name dieses Ordners wird durch den Benutzer gewählt (siehe Kapitel (g)) Zu diesem Ordner existiert eine XML-Datei, die das Projekt im ganzen verlinkt und die in diesem Fall Projekt.xml heißt.

Projekt.xml - Diese Datei beinhaltet alle Links zu den XML- Dateien, welche die Projektdaten genauer spezifizieren, wie zu Beispiel die einzelnen Devices des Projektes.

Ordner: Projekt - Durch diesen Ordner werden die projektglobalen Daten gekapselt.

Projekt.xml - Diese „Linkdatei" verweist auf die Devices und alle XML- Datei, die projektglobale Daten enthalten.

Alarms.xml - Zum Speichern der Alarmdaten eines SIMOTION SCOUT-Projektes ist diese Datei entstanden.

Installation.xml - In dieser Datei sind Komponenten von SIMOTION SCOUT beschrieben, die beim Zeitpunkt des Exports installiert waren.

ReferenceTable.xml - Diese Datei enthält alle Referenzen und Verweise, die einzelne

Teilkomponenten untereinander besitzen. Damit nach dem Import die Funktionalität der Objekte wieder vorhanden ist, wird diese Datei ausgewertet.

S7ProjectData.xml - Die Hardwarekonfiguration der Step7-Anteile wird in der Datei beschrieben.

Ordner: Device - In diesem Ordner sind alle XML- Dokumente, XSL- Files, HTML-Seiten und Unterordner enthalten, die Einzelheiten eines Device beinhalten. Da ein Projekt aus mehren Devices aufgebaut sein kann, dass mehrere Ordner dieser Art vorhanden sind.
Auch in diesem Ordner wurde eine XML- Datei erstellt, die dieses Device im XML- Format beschreibt.

Device.xml - Wie oben erklärt, verwaltet diese Datei Links zu anderen untergeordneten XML- Dateien.

RunTimeLevels.xml - In der Datei stehen die Tasks, die von dieser Device bearbeitet werden können.

DeviceData.xml - Diese Datei beinhaltet die Eigenschaften eines Device.

Device_Symbols.xml - Hier werden die Symbole und Variablen eines Device gespeichert, wie z.B. IO- Variablen.

Ordner: ProcessingUnit - Dieser Ordner enthält alle Programme, die in die CPU geladen werden können und ausgeführt werden können.

ProcessingUnit.xml - Diese Datei enthält alle Links auf die erstellten Programme.

SCL-Programm.xml - In dieser Datei wird ein Programm (MCC- Programm oder St-Programm) beschrieben.

Ordner: TOCamType - Der Ordner enthält für die Kurvenscheiben spezifische Daten.

TOCamType.xml - Alle Links auf die im Projekt enthaltenen Kurvenscheiben sind in dieser Datei enthalten.

Kurvenscheibe.xml - Hier werden die Daten einer Kurvenscheibe gespeichert.

Ordner: TOPosAxis - Dieser Order enthält alle Daten der Positionierungsachsen. Alle anderen Objekte, die in einer Station enthalten sein können, werden in einer ähnlichen Hierarchie hinterlegt, dabei macht eine Kurvenscheibe eine Ausnahme. So werden zum Beispiel alle Messtaster einer Station in einem Ordner namens „TOMeasuringInputType" gespeichert. Dieser Ordner beinhaltet eine gleichnamige XML- Datei die Links zu jeder XML-Datei eines einzelnen Messtasters. Diese XML- Files tragen den Namen des Messtasters und liegen in einem gleichnamigen Unterordner des „TO-MeasuringInputType"- Ordners. In diesem Ordner ist außerdem die Expertenliste des Messtasters im XML-Format gespeichert.

TOPosAxis.xml - Die Datei beinhaltet Links zu den einzelnen Positionierungsachsen

Ordner: Positionierungsachse - Dieser Ordner enthält alle Daten einer Positionierungsachse

Positionierungsachse.xml - Diese „Linkdatei" zeigt auf die „Datendatei" der Positionierachse.

ExpertList.xml - Die Expertenliste der Positionierungsachse ist in der Datei hinterlegt.

Patentansprüche

1. Verfahren zur Ablage und zur Bearbeitung von Projekt- und/oder Projektierungsdaten einer Automatisierungskomponente,
gekennzeichnet durch eine Untermenge folgender Merkmale:
- Bereitstellung von Projektdaten einer Automatisierungskomponente auf Basis XML,
- Export/Import von Projektdaten in XML und
- Verarbeitung der XML-Daten mit Standardwerkzeugen.

2. Verfahren nach Anspruch 1,
dadurch gekennzeichnet, dass als Standardwerzeuge eine Untermenge von
- Versionsverwaltungstools
- Externe CAD-Programme
- Externe Kurvenscheibenwerkzeuge
- Externe Productivity Tools
- Externe Skripting Mechanismen
verwendet wird.

3. System zur Ablage und zur Bearbeitung von Projekt- und/oder Projektierungsdaten einer Automatisierungskomponente,
gekennzeichnet durch eine Untermenge folgender Merkmale:
- Bereitstellung von Projektdaten einer Automatisierungskomponente auf Basis XML,
- Export/Import von Projektdaten in XML und
- Verarbeitung der XML-Daten mit Standardwerkzeugen.

4. System nach Anspruch 3,
dadurch gekennzeichnet, dass als Standardwerzeuge eine Untermenge von
- Versionsverwaltungstools
- Externe CAD-Programme
- Externe Kurvenscheibenwerkzeuge
- Externe Productivity Tools

- Externe Skripting Mechanismen verwendet wird.